UNIVERSIDAD CARLOS III DE MADRID

Escuela Politécnica Superior - Leganés

INGENIERÍA DE TELECOMUNICACIÓN

Proyecto Fin de Carrera

Plataforma Web de Simulación Remota en un *Cluster* de Computación Científica

AUTOR: Cristina García Muñoz
DIRECTOR: Adrián Amor Martín
TUTOR: Luis Emilio García Castillo

Leganés, 2014

Título:	*Plataforma Web de Simulación Remota en un Cluster de Computación Científica*
Autor:	Cristina García Muñoz
Director:	Adrián Amor Martín
Tutor:	Luis E. García Castillo

La defensa del presente Proyecto Fin de Carrera se realizó el día 3 de Octubre de 2014, siendo calificada por el siguiente tribunal:

Presidente:	Alejandro García Lampérez
Secretario:	Sergio Llorente Romano
Vocal:	Florina Almenarez Mendoza

Habiendo obtenido la siguiente calificación:

Calificación:

Presidente	**Secretario**	**Vocal**

A mi madre

...siempre habrá, estemos donde estemos,
una gran línea recta entre tu cuerpo y mi cuerpo...

Agradecimientos

Este proyecto ha sido un largo camino recorrido de la mano de muchas personas que han contribuido a que pudiera sacarlo adelante.

En primer lugar quiero dar las gracias a mi director de proyecto, pero sobre todo mi amigo, Adrián, por toda la ayuda que me ha prestado a lo largo de los últimos años sin la cual no podría haber terminado y por ofrecerme la oportunidad de trabajar con él en este departamento. También agradecer su ayuda a Nacho y a mi tutor de proyecto, Luis Emilio, piezas fundamentales en la realización de este trabajo.

Quisiera agradecer a todos mis amigos la paciencia que han tenido conmigo estos últimos meses: a mis supernenas, Sandra y Sara, porque somos un equipo inseparable estemos donde estemos; a Vero por preocuparse por mí y escucharme siempre; a Paloma porque siempre ha creido en mí como la que más y a mis compañeros de universidad por hacer de la carrera un camino mucho más llevadero y divertido.

A Rodri, porque la mitad del proyecto debería llevar su nombre, ha conseguido que llegue a buen puerto recordándome cuál era la meta y haciéndome ver que cada vez estaba más cerca sin permitir que tirase la toalla.

Por último quiero agradecer a mi familia, en especial a mi hermana y a mi padre, por darme su apoyo en todo momento haciendo que no me desanimase y siguiera adelante. Y por supuesto a mi madre, por enseñarme a ser quien soy y por motivarme a perseguir mis objetivos a pesar de las dificultades.

Resumen

La creciente demanda del uso de *clusters* de altas prestaciones, o HPCC (High Performance Computing Cluster), para la resolución de problemas computacionalmente pesados motivó la creación de la aplicación *Posidonia*. Esta aplicación, con versiones de escritorio y para Android, solucionaba la barrera de entrada que supone el envío y gestión de las tareas de forma remota, convirtiendo la simulación de trabajos en un *cluster* en una labor sencilla.

Para complementar esta aplicación y dotarla de mayor movilidad y escalabilidad, se plantea la creación de una plataforma web que implemente toda la funcionalidad de la aplicación original, haciendo uso del API (Application Programming Interface) desarrollado. En este Proyecto Fin de Carrera se describe el proceso de creación de dicha plataforma y su arquitectura, detallando la adaptación de la aplicación original y la extensión de sus funcionalidades en esta nueva versión.

Asimismo, se lleva a cabo un estudio de las distintas alternativas para desarrollar un proyecto de estas características, haciendo una comparativa de ellas y justificando la elección de *Play! framework* como herramienta de trabajo. Esta infraestructura permite reunir todas las tecnologías necesarias para la creación de aplicaciones web, proporcionando una arquitectura ordenada y simple basada en el paradigma MVC (Model View Controller) y la filosofía REST (Representational State Transfer) que facilita el trabajo al desarrollador.

Palabras clave: *user-friendly*, seguridad, movilidad, *cloud computing*, *Play! framework*, MVC, REST, ejecución remota, HPCC, supercomputación.

Abstract

The growing demand of High Performance Computing Clusters (HPCC) for heavy computing problems resolution led to the development of an application called *Posidonia*. This application, which has desktop and Android versions, gave a solution to the difficulties that users encountered when managing tasks on a remote cluster, making it simple to work in such environment.

In order to complement this application and give more scalability and mobility to it, a web site is designed introducing every feature in the original project by using its developed Application Programming Interface. For this Final Project a web site is built including a description of the complete design proccess and its architecture, detailing the adjustments made in *Posidonia* and every extension added in this new version.

In addition, a full study of the current technologies suitable for this kind of project development is made by comparing them in order to back up the selection of *Play! framework*. This framework gathers every tool needed to design web applications and provides a lightweight and ordered architecture based on the Model View Controller (MVC) pattern and Representational State Transfer (REST) model making it developer-friendly.

Key words: *user-friendly*, security, mobility, cloud computing, *Play! framework*, MVC, REST, remote execution, HPCC, supercomputing.

Índice general

Índice de figuras

Capítulo 1

Introducción

La creciente aparición de grandes problemas a nivel de capacidad computacional en la actualidad hace necesario el acceso a entornos de supercomputación para su resolución. La principal ventaja que tienen este tipo de infraestructuras frente a ordenadores de sobremesa es que su configuración está optimizada para el tratamiento de estos problemas de manera que el tiempo de ejecución se reduce en varios órdenes de magnitud. Es por ello que en entornos de ciencias e ingeniería se acuda con frecuencia a infraestructuras específicas de simulación o *clusters* de alto rendimiento, HPCC (High Performance Computing Cluster). Sin embargo, el acceso a estas tecnologías tiene una curva de aprendizaje elevada ya que se requiere conocimiento del *software* y del *hardware* de los equipos así como de los protocolos de transferencia de ficheros y control remoto de los recursos. Además, todas estas herramientas de supercomputación están pensadas para ser utilizadas al mismo tiempo por múltiples usuarios, lo que añade la necesidad de tener mecanismos de gestión de colas para que no se produzcan interferencias entre usuarios, aumentando los requisitos necesarios para manejar su uso. Para dar solución a este escenario de partida que se muestra en la Figura 1.1 surgió la aplicación *Posidonia*, [5], que ofrece una respuesta a la necesidad descrita con las siguientes características: orientación a usuario, movilidad, seguridad, eficiencia, extensibilidad y generalidad.

Sin embargo, la movilidad que proporciona la aplicación, que cuenta con una versión de escritorio y una versión para dispositivos móviles programada en Android, [6], aún podría ser mejorada para dar acceso a este servicio desde cualquier dispositivo con acceso a Internet sin necesidad de tener la aplicación instalada. Para ello se propone el desarrollo de una plataforma Web que implemente toda la funcionalidad contenida en la aplicación original desde cualquier navegador incluidos los dispositivos móviles, optimizando de esta forma la movilidad e incluyendo, además, nuevas funcionalidades.

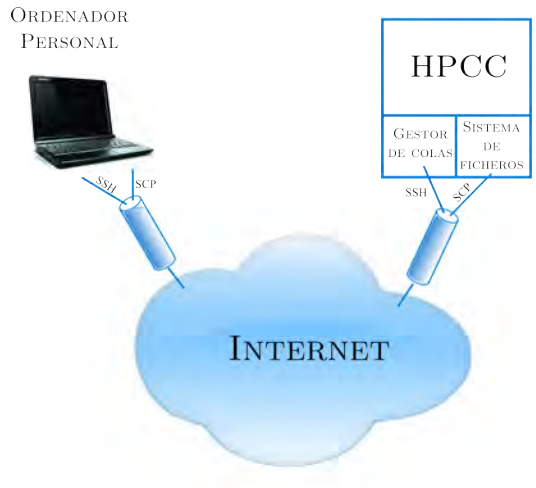


Figura 1.1: Escenario inicial de acceso a HPCC. Fuente: [1].

Para este desarrollo se produce un cambio en el escenario de partida, como se ve en la Figura 1.2, donde se introduce un servidor intermedio que se encarga del procesamiento evitando la carga en los dispositivos, donde sólo se muestran los resultados. De esta manera se consigue una aplicación cuyo acceso está disponible incluso para dispositivos con limitaciones en *hardware*.

Para potenciar la movilidad de la aplicación *Posidonia* se dispone de una plataforma Web cuyos requisitos para poder utilizar en modo de producción son los siguientes:

- Movilidad: ya era uno de los objetivos de *Posidonia*, que llevó a su implementación en dos formatos, herramienta de escritorio y como aplicación de Android para tabletas y dispositivos móviles. Con el fin de abarcar todo el espectro, se pretende llegar a cualquier sistema operativo y navegador de modo que la plataforma sea funcional desde cualquier punto de acceso a Internet.

- Escalabilidad: para que *Posidonia* pueda afrontar el crecimiento en el número de usuarios es imprescindible que sea escalable de manera sencilla. Además, se debe poder adaptar a modificaciones y futuras actualizaciones sin tener que alterar drásticamente la estructura original. En relación a las actualizaciones cabe destacar la gran ventaja que supone tener una Web frente a las aplicaciones de escritorio o Android. En el caso de estas dos últimas es necesario proporcionar a los usuarios un sistema de actualización de sus versiones, con los consiguientes problemas de compatibilidad entre los usuarios

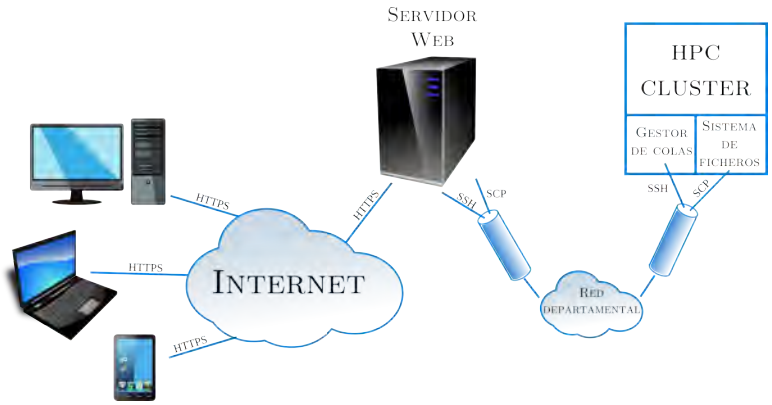


Figura 1.2: Escenario para el desarrollo de la plataforma Web.

con versiones antiguas. Con un servicio Web se consigue centralizar en un único punto, el servidor, las modificaciones de forma que los usuarios acceden siempre a la última de las versiones. Por último se propone como avance el almacenamiento de los datos en el servidor en una base de datos, más eficiente que el sistema de ficheros de *Posidonia* para grandes cantidades de información.

A continuación se va a hacer un recorrido por la estructura y funcionalidad de la aplicación de base, *Posidonia*, para que se tenga una visión clara de cuál es el punto de partida.

El escenario de la Figura 1.1 está pensado para un ordenador personal con recursos computacionales limitados desde el que se pretende realizar tareas de simulación en un *cluster* sirviéndose de una conexión a Internet. Para mantener la seguridad y confidencialidad en este proceso es necesario que la comunicación se haga sobre los protocolos de seguridad de tranferencia de archivos y comunicación entre sistemas, SCP (Secure CoPy, [7]) y SSH (Secure SHell, [8]). Además hay que asegurar el acceso a esta funcionalidad desde cualquier dispositivo y sistema operativo por lo que la aplicación está desarrollada completamente en Java que es multiplataforma.

La funcionalidad de la aplicación *Posidonia* se puede resumir en los siguientes puntos.

- Envío de trabajos: es la funcionalidad básica, que permite al usuario seleccionar los archivos que necesita mandar al *cluster* para ejecutar su simulación de manera sencilla y rápida.

- Configuración de las opciones de ejecución: en caso de estar familiarizado con los *clusters* de alto rendimiento, esta funcionalidad permite al usuario cambiar una serie de parámetros avanzados del trabajo que va a enviar como la memoria virtual a utilizar.

- Control y monitorización de las tareas en ejecución: cuando un trabajo es enviado, se permite a su propietario hacer un seguimiento del estado de este e interrumpir su ejecución si lo desea antes de finalizar.

- Notificación de trabajos finalizados: cuando una tarea finaliza su ejecución mientras la aplicación está activa, se genera una notificación mostrada al usuario y se descargan los ficheros de salida de manera automática. En caso de que la tarea finalice estando la aplicación inactiva, se mostrará una notificación al reinicio de la misma que indique todos los trabajos que han acabado su ejecución mientras *Posidonia* no estaba ejecutándose.

- Histórico de trabajos: cada usuario dispone del acceso a un historial de todas las tareas que ha mandado al *cluster* que le permite seleccionar trabajos y descargar sus archivos de entrada o los resultados que generó su ejecución. También se permite borrar elementos de este historial, lo que supone que se elimina cualquier rastro de la tarea en el propio *cluster*. Esta funcionalidad contribuye a mejorar uno de los requisitos que la aplicación busca satisfacer, la movilidad, ya que actúa como repositorio de todos los trabajos del usuario y se puede acceder a él desde cualquiera de las versiones de *Posidonia*.

También es importante conocer la arquitectura de la aplicación para determinar la del proyecto Web a desarrollar ya que va a tener las mismas características. La estructura del código desarrollado se muestra en la Figura 1.3, donde se ve la comunicación entre el ordenador personal y el *cluster*. La aplicación tiene tres partes diferenciadas: la interfaz gráfica, el protocolo de comunicación y la ejecución de trabajos en el *cluster*, que se detallan a continuación.

La ejecución de trabajos en el *cluster* se divide en tres capas, formando la pila mostrada en la Figura 1.4. Ya se mencionó que el *cluster* debe ser accedido haciendo uso del sistema de colas SGE [9] (Sun Grid Engine, ahora llamado OGE, o Oracle Grid Engine) u otro similar como HTCondor [10], que constituye la primera de las capas. Para poder ejecutar trabajos usando Java se utiliza una plataforma llamada DRMAA [11] (Distributed Resource Management Application API), que permite el control de los trabajos en el gestor de colas utilizado y se establece como capa intermedia de esta pila de ejecución. Por último, se implementa un API (Application Programming Interface) intermedio, llamado `ClusterJob`, para corregir los contratiempos que aparecen del uso de DRMAA.

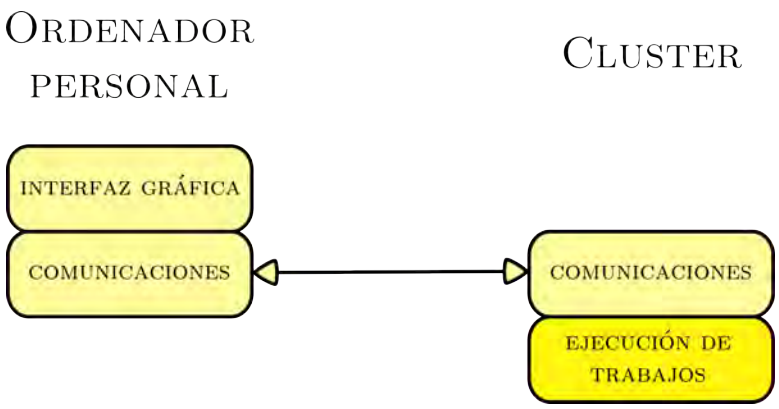


Figura 1.3: Estructura simplificada del código de *Posidonia*.

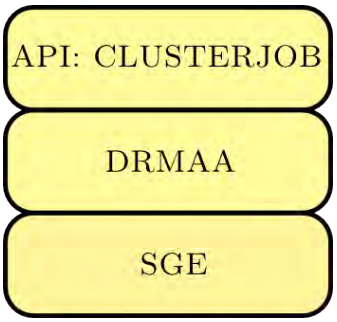


Figura 1.4: Capas de la ejecución de trabajos en un *cluster* mediante *Posidonia*.

El uso de SGE o HTCondor como sistema de colas permite planificar y otorgar recursos a los trabajos que están siendo ejecutados en el *cluster* de manera flexible, haciendo posible asignar distinto número de procesadores, memoria y otros recursos a diferentes trabajos a la vez.

DRMAA es un API desarrollado con el objetivo de controlar y enviar trabajos a uno o más sistemas DRM (Distributed Resource Management) que actúan como planificador de tareas en un *cluster*. Las ventajas de utilizarlo son:

- Permite ejecutar trabajos de un gestor de colas desde Java, ofreciendo una interfaz de alto nivel para cualquier aplicación que necesite gestionar un sistema de colas, también conocido como *scheduler*. Todo lo que se puede hacer mediante comandos nativos del gestor de colas utilizado puede hacerse a través de este API.
- Independencia del gestor de colas empleado: proporciona una capa de abstracción sobre el *scheduler* utilizado, haciendo posible cambiar la capa inferior por otro sistema sin modificar la estructura de la aplicación. Gracias

a esta independencia se ha podido hacer una migración a otro sistema, HTCondor [10], del que se habla en el capítulo 3.

- Captura del evento de finalización de trabajo: en los gestores de colas es frecuente que el mecanismo de notificación del final de una tarea sea el envío de un correo al usuario; sin embargo, DRMAA es capaz de capturar este evento de finalización, posibilitando una manera más elegante de avisar al usuario instantáneamente del fin de la ejecución evitando esperas activas en el código.

A pesar de ser un sistema idóneo para desarrollar la funcionalidad de *Posidonia*, hay una serie de inconvenientes que han de tenerse en cuenta. El proyecto DRMAA, aunque ambicioso, está estancado en su versión de Java de manera que no se cuenta con una documentación completa y rigurosa de su utilización, la corrección de fallos o extensión del API no está disponible. Para solucionar los problemas que surgen de esta situación se desarrolla el API `ClusterJob` que forma la capa superior de esta pila de ejecución. Sus beneficios respecto a la capa intermedia son:

- Contiene una documentación completa, correcta y detallada de la toda la funcionalidad que ofrece el API.
- Gestión completa de los ficheros de error y de salida, pudiéndose sobreescribir en caso de que ya exista, corrigiendo así el funcionamiento de DRMAA.
- Uso del API estilo Java: para el usuario es transparente el uso de métodos de la capa inferior.
- Estructura de métodos simples, de tipo `get` y `set` cumpliendo el principio de encapsulamiento característico de Java para consultar y modificar los parámetros de ejecución de la tarea, y `runJob` para ejecutar el trabajo con dichas condiciones.

El protocolo de comunicación entre el ordenador personal y el *cluster* debe permitir la transferencia de archivos e instrucciones a través de Internet, garantizando los siguientes requisitos:

- Orientación a sesión, no a conexión. Como toda la información sobre los trabajos enviados está en remoto, es decir, en el *cluster*, es indiferente que el usuario tenga la aplicación activa en su ordenador personal. El carácter de estos trabajos que pueden tardar incluso días en ejecutar, hace que sea necesario un protocolo orientado a sesión y no a conexión.

- Eficiencia: se intenta enviar el menor número de mensajes para minimizar el uso de la red, lo que favorece tanto a la velocidad de la conexión del usuario como al *cluster*, que puede atender múltiples peticiones a la vez.
- Seguridad: el protocolo diseñado emplea SSH y SCP para implementar un nivel de seguridad adecuado. Además se debe garantizar la confidencialidad de los datos transferidos entre cliente y servidor por lo que se implementa su encriptación para que sean entendidos únicamente por las entidades involucradas. Para ello se utiliza la librería JSch (Java Secure channel, [12]) que es una implementación pura del protocolo SCP en Java.
- Acceso concurrente a los ficheros empleados: hay tres archivos necesarios para dar la funcionalidad a la aplicación y que se almacenan encriptados en el *cluster* para cada usuario. Estos ficheros son: `runningJobs`, con la información de los trabajos en curso, `history`, que almacena las tareas que ya han terminado y `pendingJobs`, que informa de los trabajos cuya ejecución terminó mientras la aplicación ha estado desconectada.

El protocolo de comunicaciones es resistente a caídas de red y pérdidas de conexión, característica clave para funcionar en el escenario propuesto. Para cumplir todos los requisitos el protocolo de comunicaciones cuenta con tres fases que se muestran esquematizadas en la Figura 1.5.

1. Subida de ficheros al *cluster*: se envían, a través de un canal seguro, tanto los archivos que el usuario selecciona como parte del trabajo enviado, como los generados por la aplicación que contienen datos sobre la configuración de la tarea enviada y de la sesión abierta por el usuario.
2. Ejecución de la tarea: en función de los parámetros contenidos en los archivos de configuración enviados, el API `ClusterJob` se encarga de ejecutarla en el *cluster*.
3. Fin de ejecución de la tarea y descarga de resultados: una vez recogido el evento DRMAA de finalización de la tarea, se procesa para proceder a la descarga automática de los resultados en caso de que el usuario tenga la aplicación activa. Esto se consigue enviando un mensaje de notificación a la herramienta que debe tener un servidor TCP levantado en un puerto libre. De la misma forma, si el trabajo se interrumpe antes de finalizar, se notifica al usuario con el mensaje correspondiente.

Por último, cabe mencionar el esquema de clases Java desarrolladas, diferenciando entre las que pertenecen al equipo local y las que se encuentran alojadas en el

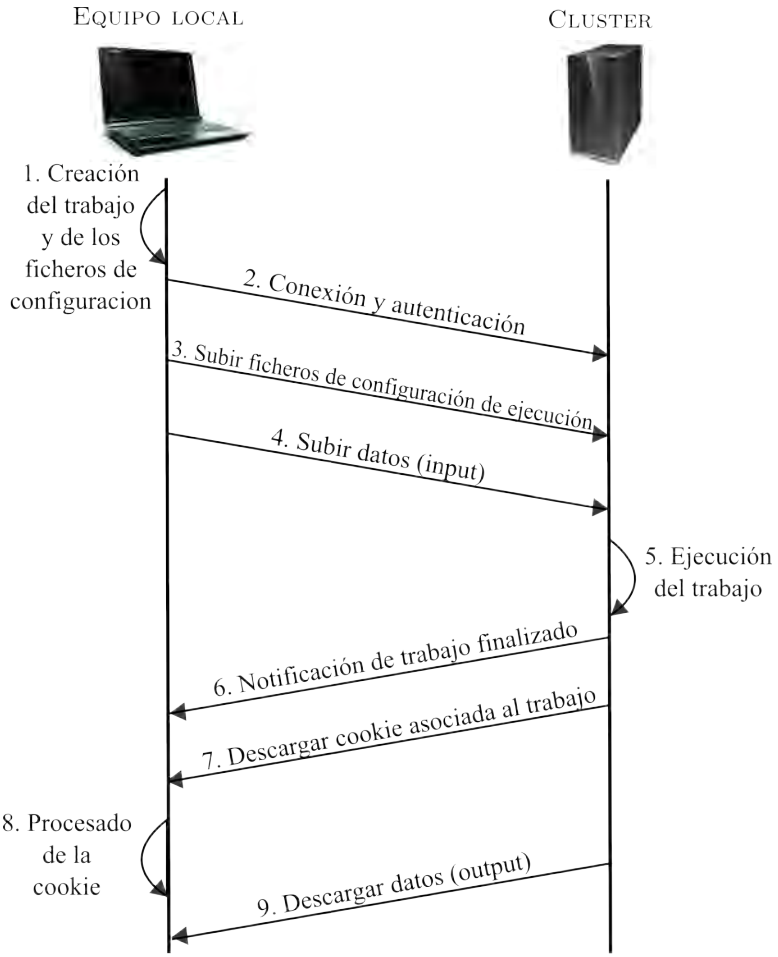


Figura 1.5: Protocolo de comunicaciones en la aplicación de escritorio.

cluster. Esto es necesario ya que la aplicación Web a desarrollar se sitúa sobre la capa de comunicaciones en el equipo local que, en caso de la plataforma Web, sería el servidor de la misma. La Figura 1.6 muestra los paquetes que la herramienta necesita para implementar el protocolo anteriormente descrito:

- Paquete `Connections`: se encuentra en el equipo local, por lo que se encarga de ofrecer la funcionalidad del protocolo colaborando con `SSHCondor`, es decir, gestiona la transferencia de archivos al *cluster* y ejecuta los comandos necesarios. Cuenta con las siguientes clases:

  1. `ClusterConnection`: constituye el núcleo del protocolo con un conjunto de métodos que implementan la totalidad de los servicios que ofrece la capa de comunicaciones.

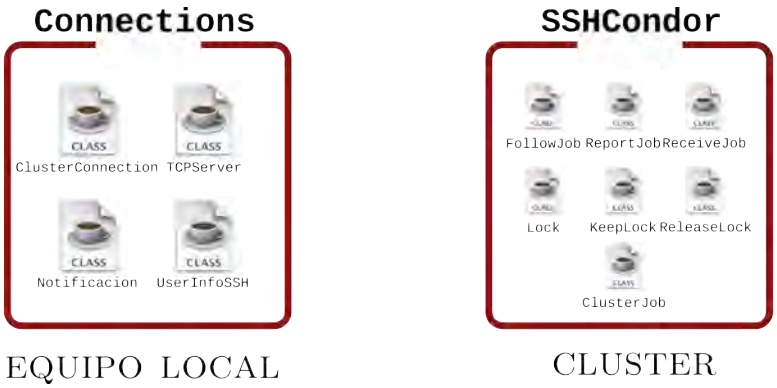


Figura 1.6: Esquema general de la estructura de clases del protocolo de comunicaciones.

2. `TCPServer`: se encarga de implementar el servidor TCP, imprescindible para recibir las notificaciones del *cluster* tras los eventos de finalización e interrupción de trabajos. Además incluye el código de respuesta a la llegada de notificaciones y un método para la búsqueda de un puerto libre ya que al no ser una aplicación registrada no se puede fijar un puerto concreto.
3. `Notification`: es una clase que controla y define las notificaciones de las que hace uso *Posidonia*.
4. `UserInfoSSH`: necesaria para la implementación de las conexiones JSch.

- Paquete `SSHCondor`: radica en el *cluster* e incluye el API desarrollada para la parte de ejecución de trabajos. Su función es, principalmente, el envío de trabajos al gestor de colas y la modificación de los archivos `runningJobs`, `pendingJobs` y `history`. Además, incorpora las clases de Java que garantizan el acceso concurrente a estos ficheros.

La tercera y última de las capas corresponde a la interfaz gráfica de la aplicación. En función de la versión que se utilice, escritorio o Android, existe una interfaz gráfica adaptada a cada una de ellas; sin embargo, hay una serie de características que comparten ambas modalidades:

- Inteligencia limitada: para garantizar la extensibilidad de la aplicación es importante que toda la inteligencia esté en capas inferiores de manera que la interfaz aporte diseño y no funcionalidad.

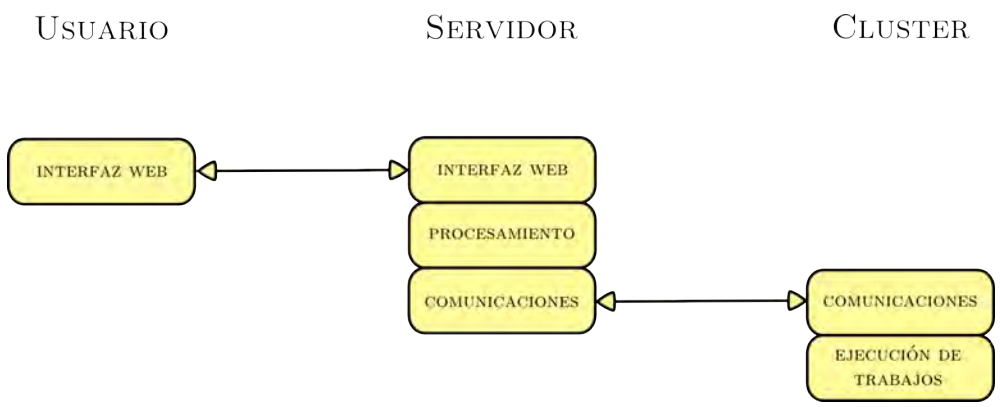


Figura 1.7: Esquema de la plataforma Web.

- *User-friendly*: la interacción con el usuario debe ser intuitiva y sencilla contando con una ventana principal simple. Por ello, se ocultan los parámetros avanzados en una ventana aparte, para los usuarios con el suficiente conocimiento para manejarlos.

- Presentación clara de los datos: toda la información que se obtiene de la capa de comunicaciones debe ser mostrada con la información relevante para el usuario. La forma de presentación elegida es una tabla para todas las versiones, tanto para los trabajos aún en ejecución como para el historial de trabajos ya finalizados.

- Notificación de lo más relevante: la aplicación muestra avisos únicamente cuando es necesario, es decir:

    1. Cuando el trabajo ha llegado al gestor de colas indicando al usuario que el envío de la tarea al *cluster* se ha realizado de manera satisfactoria.
    2. Al finalizar la ejecución, que marca la descarga automática de los resultados.
    3. Cuando se termina la descarga de los ficheros solicitados, ya sean de entrada o de salida, para que el usuario pueda consultarlos.

Una vez explicada la aplicación *Posidonia* se puede situar este Proyecto Fin de Carrera. Se trata de modificar la interfaz que pasa de ser de escritorio o una aplicación Android a ser una interfaz Web. Además, se añade inteligencia al servidor Web, donde se hace todo el procesado de la información y su persistencia en la base de datos y se incorpora la comunicación con el cluster. El esquema pasa a ser el que se muestra en la Figura 1.7.

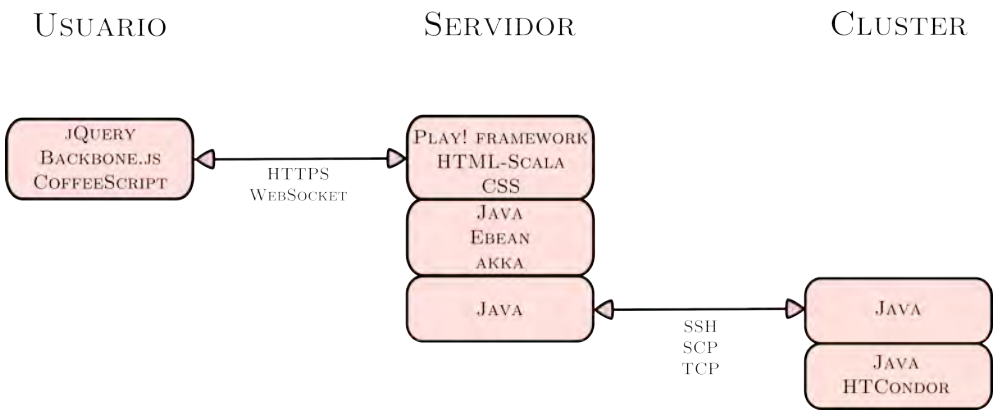


Figura 1.8: Esquema de tecnologías utilizadas en la plataforma Web.

Para cada uno de los componentes de esta pila de protocolos se utilizan las tecnologías que más se adecúan a las características de la aplicación a desarrollar. Estas quedan resumidas en la Figura 1.8 y serán detalladas en el Capítulo 3.

Los objetivos de este Proyecto Fin de Carrera que serán desarrollados a lo largo de esta memoria son los siguientes:

- Diseño e implementación de una plataforma Web con todas las funcionalidades de *Posidonia* explicadas en este capítulo:

  1. Envío de tareas al *cluster*
  2. Consulta de las tareas finalizadas (historial) y en ejecución
  3. Eliminación de tareas del historial y cancelación de tareas en ejecución
  4. Descarga de los archivos de entrada y salida de cada trabajo

  Además, la plataforma debe tener las funciones típicas de cualquier página Web con acceso a una zona privada, como la autenticación, *log-out* y registro de usuarios.

- Ampliación de la interfaz de usuario: con el objetivo de mejorar la aplicación original se añaden una serie de funcionalidades a incorporar en la página Web:

  1. Orientación a usuario Web: en la aplicación original era necesario hacer una conexión (inicio de sesión) para cada uno de los servidores a los que se tiene acceso. La Web debe permitir que cada uno de sus usuarios pueda registrar todos los servidores a los que tiene acceso introduciendo una única vez sus credenciales de forma que se gestionan de manera tranparente para el usuario que cuando accede a la Web

tiene disponibles todos sus trabajos en cada uno de los servidores que ha introducido.

2. Administrador del sitio Web: para facilitar un mantenimiento básico de la Web, es necesario habilitar una serie de funciones de acceso único para los administradores de la Web, que les permitan realizar tareas como borrar y añadir servidores o aplicaciones disponibles y actualizar datos que de otra forma requerirían la modificación directamente de la base de datos o de la propia aplicación.
3. Interfaz para dispositivos móviles: para complementar la versión en Android y, además, suplir la carencia de versiones para otros sistemas operativos, es conveniente hacer una versión simplificada de la interfaz en caso de que la conexión se haga a través de una *tablet* o *smartphone*.
4. Compartición de trabajos: los usuarios pueden compartir trabajos con otros usuarios de la Web con sólo introducir la dirección de correo que identifique al otro usuario en la plataforma. El beneficiario de la compartición tendrá acceso a la descarga de archivos de entrada y salida pero no podrá borrar los trabajos del sistema ni compartirlo con otros usuarios.

Capítulo 2

Tecnologías para el desarrollo Web

Este capítulo sirve como recorrido del estudio realizado sobre distintas alternativas para llevar a cabo el desarrollo de una plataforma Web que implemente la funcionalidad de *Posidonia*. El primer paso para este estudio es analizar qué requisitos tiene que satisfacer el diseño de una plataforma Web de las características presentadas en el capítulo anterior. Los elementos imprescindibles son los siguientes:

- Prestaciones: tiempos de respuesta e interacción eficiente entre componentes.
- Escalabilidad: posibilidad de incorporar nuevos servidores de la forma más sencilla posible.
- Disponibilidad: seguridad frente a caídas y tolerancia a fallos.
- Independencia de plataforma: se pretende que las aplicaciones puedan ser ejecutadas en cualquier entorno a nivel de servidor (Linux, Windows) o a nivel de cliente (navegador), sin tener que añadir modificaciones.
- Almacenamiento y acceso de datos: empleo de sistemas de bases de datos que permitan la persistencia de la información en el sistema.
- Consistencia de los datos: control de acceso concurrente a la información, de forma que los datos a los que se accede sean siempre correctos y actualizados.
- Interacción con el usuario: que incluye la autenticación y coordinación de accesos concurrentes.

- Mantenibilidad y portabilidad.

La arquitectura clásica para este tipo de aplicaciones sigue el modelo cliente servidor, en el que ambos están débilmente acoplados y se comunican mediante mensajes. Dependiendo de la función que desempeñe el cliente puede ser 'gordo' o 'pesado', si la parte principal de la aplicación se ubica en el cliente, o 'delgado', si por el contrario se encuentra en el servidor. En general, para aplicaciones Web, se tiene un cliente delgado que inicia la comunicación a través de peticiones y un servidor pasivo encargado del procesado. Las ventajas de este modelo van desde una menor infraestructura en el lado del cliente que simplifica el sistema hasta una administración más fácil y una disminución del tráfico en la red ya que el procesado se concentra en el servidor. Además, se posibilita una gestión centralizada de los recursos que permite ganar en seguridad, integridad de los datos y control sobre las transacciones.

Dentro de este modelo, las arquitecturas multi-nivel son aquellas en las que se añaden niveles intermedios que llevan a cabo tareas críticas. De esta forma se logra extender la funcionalidad del lado del servidor de forma sostenible y se consigue una separación con interfaces de comunicación claramente diferenciadas. Las ventajas de este tipo de arquitecturas extienden a las ya mencionadas de tener un cliente delgado: dota al sistema de mayor flexibilidad y escalabilidad, un control más fino de la carga del servidor para proporcionar tiempos de respuesta más bajos y la capacidad de detectar y depurar errores debido a la modularidad. Un ejemplo de arquitectura multi-nivel de tres niveles se muestra en la Figura 2.1 donde podemos ver un cliente, un nivel medio y un nivel de datos. Para extender la arquitectura a una con más niveles basta con extender el nivel intermedio, que se puede separar en presentación y procesado y que constituye una de las claves para un desarrollo de aplicaciones ordenado y eficiente.

El primer nivel o nivel cliente es la parte de la aplicación que se ejecuta en la máquina del cliente cuyas funciones se limitan a mostrar la información del servidor y recoger datos de entrada. Típicamente se trata de un navegador Web que incluye applets, para mostrar la información gráfica, código JavaScript [13] para pre-procesar las entradas y plugins para características adicionales. El nivel medio está constituido por la lógica de presentación y la lógica de negocio (también conocida como capa de servicios empresariales). La primera recibe las peticiones de los clientes y extrae la información, llama a los métodos de la capa de servicios empresariales necesarios y prepara las respuestas a esas peticiones. La lógica de negocio es la parte principal de la aplicación que incluye la funcionalidad y se encarga de preparar la información del usuario. Por último, la capa de datos se encarga de la persistencia de la información en la aplicación.

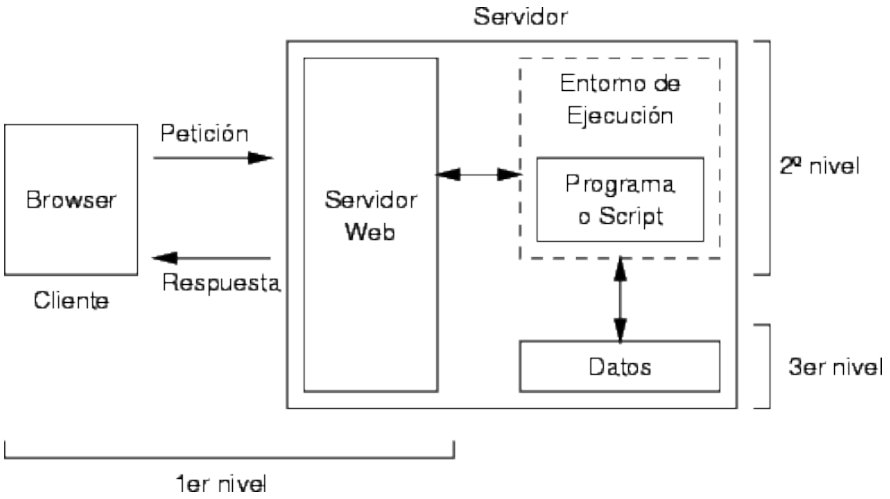


Figura 2.1: Arquitectura multi-nivel. Fuente: [2].

2.1. Tecnologías

2.1.1. HTML

HTML (HyperText Markup Language, [14]) es el lenguaje predominante para la creación de páginas Web. Se trata de documentos escritos en texto plano con la extensión .html o .htm que describe el significado de distintos elementos. Su funcionamiento está basado en el uso de etiquetas encerradas entre corchetes angulares <> que no se muestran en la página pero ofrecen las instrucciones necesarias al navegador sobre el significado del texto. Es importante destacar que HTML no está diseñado para describir la apariencia de la página por lo que conviene separar apariencia de contenido en todo momento. Las etiquetas aparecen siempre por parejas, una de apertura y otra de cierre que generalmente es igual pero con el carácter / delante. Pueden estar incrustadas unas en otras para indicar la organización de la página, es decir, aparecer anidadas, pero es importante cerrarlas en el orden inverso de la apertura.

El marcado HTML tiene una serie de componentes imprescindibles: elementos y atributos. Los elementos son el componente estructural básico de HTML y pueden tener, a su vez, atributos y contenido. Normalmente están compuestos por un par de etiquetas, de inicio y cierre, que delimitan el contenido, mientras que los atributos están dentro de la etiqueta de inicio. Este marcado estructural describe el propósito del texto dentro del conjunto y no su posición o apariencia. Existe un marcado presentacional que sí describe la apariencia del texto en lugar de su funcionalidad; sin embargo, este marcado es bastante limitado y no permite

Fichero 2.1: Etiquetas minimas en HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Ejemplo1</title>
  </head>
  <body>
    <p>ejemplo1</p>
  </body>
</html>
```

personalizar en gran medida diseño de los documentos. Por ello para el caso que atañe al desarrollo de una aplicación Web se prefiere obviar este marcado presentacional y sustituirlo por páginas de estilo, CSS (Cascading Style Sheets, [15]), que se explicarán en el siguiente apartado. Por último existe un marcado hipertextual que se usa para enlazar partes del documento con otras partes del mismo o con enlaces externos.

Los atributos son, en general, pares de nombre-valor, que se incorporan dentro de algunas etiquetas modificando el funcionamiento de las mismas o del contenido del elemento al que afectan. Las etiquetas mínimas de un documento HTML sencillo son las que se muestran en el Fichero 2.1.

HTML5, [16], es el nuevo estándar de HTML en el que se trabaja desde el año 2011. Aunque todavía se encuentra en fase experimental, la mayoría de los navegadores lo soportan debido a las ventajas que supone. Las novedades de este estándar incluyen:

- Nuevas características basadas en HTML, CSS, DOM (Document Object Model, [17]) y JavaScript.
- Se reduce la utilización de plugins externos como Adobe Flash [18].
- Mejora del manejo de errores.
- Aumento del marcado para reemplazar los *scripts*.
- Independencia de los dispositivos.
- Proceso de desarrollo abierto al público.

Uno de los cambios más importantes es la reducción de la declaración DOCTYPE, encargada de definir el tipo de documento, que queda de la siguiente forma:

```
<!-- En HTML -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/
    strict.dtd">

<!-- En HTML5 -->
<!DOCTYPE html>
```

De las características añadidas en HTML5 las más importantes son:

- Incorporación del elemento `<canvas>` para dibujar en dos dimensiones.
- Incorporación de los elementos `<video>` y `<audio>` para la reproducción de contenido multimedia.
- Nuevos elementos específicos para cada contenido `<nav>`, `<footer>`, `<header>`, etc. Similares a algunos elementos ya existentes pero con significado semántico.
- Almacenamiento local.
- Nuevos controles: hora, calendario, URL, búsquedas, etc.
- Mejora de los formularios: añadiendo atributos que permiten controlar el formato de sus campos.

2.1.2. CSS

CSS (Cascading Style Sheets, [15]) es una herramienta que permite definir la apariencia de una página Web. Aunque se puede hacer un diseño básico de páginas Web utilizando HTML, los resultados son bastante limitados y estéticamente deficientes a pesar de las múltiples etiquetas que se fueron añadiendo para intentar evolucionar este aspecto. CSS permite embellecer las páginas generadas con HTML de una manera sencilla puesto que cuenta con infinidad de elementos para personalizarlas. Con CSS se puede, por ejemplo, cambiar colores de las imágenes, añadir fondos y bordes, cambiar la apariencia de listas, tablas, links, etc. Además, permite simplificar mucho los documentos HTML ya que todo el formato se almacena en las páginas de estilo. El hecho de que el formato se codifique aparte posibilita, además, la reutilización del mismo para distintas páginas HTML.

Para escribir CSS dentro de un documento HTML es necesario el uso de la etiqueta HTML `<style>` que permite interpretar el texto encerrado entre esta etiqueta

Fichero 2.2: Ejemplo de CSS

```
<style type= text /css >
  body{
    color: yellow;
    background-color: red;
  }
  h1{
    color: red;
    background-color: yellow;
  }
</style>
```

Fichero 2.3: Ejemplo de hoja de estilo interna

```
<!DOCTYPE>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Ejemplo</title>
    <style type="text/css">
      body {
   color:green;
   background-image:-moz-linear-gradient(top, #000000, #FFFFFF);
   background-size: 1200px 600px;
   background-repeat: no-repeat;
   font-size: 5em;
   text-align: center;
      }
    </style>
  </head>
  <body>
    <p>HOLA</p>
    <p>MUNDO</p>
  </body>
</html>
```

como una hoja de estilo. En esencia, se reduce a una lista de selectores que indican los elementos dentro de la página que se quieren modificar. Cada uno de ellos contiene un determinado número de reglas de estilo aplicables a un atributo del selector. Cuando el navegador se encuentra con los elementos especificados por los selectores, aplica las reglas correspondientes para cambiar su apariencia. Un ejemplo sería el que se ve en el Fichero 2.2.

Se puede apreciar en el ejemplo cómo cada regla se compone de un nombre seguido de dos puntos y el valor asociado. Además, las reglas de cada selector van agrupadas entre corchetes y el atributo `type` de la etiqueta contiene siempre el valor `text/css`. Para comenzar a familiarizarse con las hojas de estilo conviene hacer un *HolaMundo* cuyo diseño gráfico cuente con algunos efectos básicos. En este caso se ha escrito un pequeño código que represente un texto en color sobre un fondo degradado en escala de grises. El código que consigue estas características es el que aparece en el Fichero 2.3.

El ejemplo muestra una forma sencilla de utilizar CSS para personalizar la presentación de una página sencilla prácticamente sin contenido. Sin embargo, no es la única forma de utilizar CSS para dar formato. En concreto, hay tres formas de hacer lo mismo, cada una de las cuales responde a unas necesidades distintas.

1. Estilo en línea: se inserta el lenguaje de estilo dentro de una etiqueta HTML. Aunque funciona de la misma forma no es un procedimiento adecuado ya que impide la separación real entre el diseño y el contenido además de ser un método largo y poco elegante.

2. Hojas de estilo internas: se trata de una hoja de estilo incrustada en el documento HTML. Se introduce dentro del elemento cabecera `<head>` haciendo uso de la etiqueta `<style>`, y es el método utilizado en el ejemplo anterior. Aunque conseguimos la separación entre contenido y presentación, no resulta adecuado para la reutilización de las hojas de estilo. Tiende a ser útil cuando se quiere otorgar alguna característica a una página Web en un mismo fichero.

3. Hojas de estilo externas: la hoja de estilo está almacenada en un fichero distinto al documento HTML. Es la manera más potente de trabajar porque la separación de contenido y estilo es total.

La elección de una de estas opciones corresponde al autor del documento HTML, aunque CSS también permite que el usuario que ve el documento modifique el estilo mediante un archivo CSS especificado en la configuración del navegador que sobrescribe el definido por el autor.

Las ventajas de utilizar CSS son:

- La posibilidad de llevar a cabo un control centralizado de la apariencia de una página Web permite una actualización mucho más rápida y sencilla.

- La separación de la presentación y el contenido no sólo facilita el diseño de la página Web, sino que hace posible la reutilización de hojas de estilo para distintos contenidos o la utilización de distintas hojas de estilo para el mismo contenido, lo que proporciona versatilidad. Esta reutilización permite, además, disminuir el tiempo de recarga de la página Web.

- Se evitan antiguas soluciones poco eficientes para organizar la presentación de páginas Web como las tablas que iban en perjuicio de algunas funcionalidades de los sitios Web.

CSS3, [19], es la última versión de CSS desde el año 2011 y a diferencia de la anterior versión, CSS2 [20], está dividido en módulos. Las especificaciones antiguas han sido divididas en bloques más pequeños y se han añadido otras nuevas. Entre los módulos más importantes se encuentran:

- Selectores
- Modelo de cajas
- Fondos
- Efectos de texto
- Transformaciones 2D/3D
- Animaciones
- Interfaz de usuario
- División en múltiples columnas

La modularidad permite que no todas las especificaciones estén en el mismo estado de desarrollo. De esta forma, paulatinamente son incorporadas por los navegadores más importantes permitiendo el uso progresivo de las nuevas características que se van añadiendo.

Para entender mejor en qué consiste la modularidad de CSS se utiliza a modo de ejemplo la opción de añadir y modificar bordes que se ilustra en el Fichero 2.4 para un elemento dado y tres modificaciones posibles del mismo. Con CSS3 se pueden hacer perfiles redondeados, añadirles sombreado e incluso utilizar imágenes como relleno sin la necesidad de usar un programa de edición de imágenes. Para esto es necesario conocer las propiedades de los bordes: radio `border-radix`, sombra `box-shadow` e imagen `border-image`. Para redondear las esquinas bastaría con modificar el radio. Si lo que se quiere es añadir sombra a una caja se modifica la propiedad `box-shadow` siendo los parámetros que aparecen: desplazamiento horizontal (en píxeles), desplazamiento vertical y degradado de la sombra.

2.1.3. JavaScript

En los dos apartados anteriores se ha descrito cómo se define el contenido de una página Web y cómo podemos darle forma y apariencia. Sin embargo, es necesario añadir programación para darles dinamismo y movimiento. Para lograr

Fichero 2.4: Modificacion de bordes con CSS

```
div{
  border:2px solid;
  border-radius:25px;
  -moz-border-radius:25px; /* Firefox 3.6 and earlier */

  box-shadow: 10px 10px 5px #888888;

  border-image:url(border.png) 30 30 round;
  -moz-border-image:url(border.png) 30 30 round; /* Firefox */
  -webkit-border-image:url(border.png) 30 30 round; /* Safari and Chrome */
  -o-border-image:url(border.png) 30 30 round; /* Opera */
}
```

un comportamiento interactivo de la página se debe incorporar código en el lado del cliente. En este punto entra en juego JavaScript [13] (a menudo abreviado JS), integrado en la mayoría de navegadores, que permite dotar a las páginas de una parte imprescindible de comunicación con el usuario.

JavaScript es un lenguaje de programación desarrollado en sus inicios por Netscape Communications, cuyas principales características son:

- Es un lenguaje interpretado: está basado en *scripts*, es menos estricto en comparación a otros lenguajes de programación y carece de pasos intermedios como la compilación, lo que hace que sea fácil de utilizar.
- Programación del lado del cliente: aunque existe una forma de JS para programar en el lado del servidor, en esta sección nos centramos en su utilización más común, implementado como parte del navegador para dotar a las páginas Web de una interfaz de usuario y un carácter dinámico.
- No está relacionado con Java: a pesar del parecido de sus nombres, no están relacionados, aunque si adopta convenciones de Java. Inicialmente se pensó como una simplificación para controlar applets de Java pero tienen semánticas y propósitos distintos.
- Es orientado a objetos y basado en prototipos: la creación de objetos no se basa en la instanciación de clases, sino en la clonación de otros objetos, o mediante código del programador; es decir, un objeto existente puede servirle al programador de prototipo.
- Es débilmente tipado: el tipado es dinámico, y se corresponde con el valor y no con la variable por lo que una misma variable puede ser, por ejemplo, numérica en un momento y una cadena en otro.
- Funciones: las funciones son objetos en sí mismos y pueden anidarse.

La forma de añadir código JS a un documento HTML es similar a la manera en que se incluye CSS y tiene las mismas modalidades:

- Embebido en la cabecera: haciendo uso de la etiqueta `<style>` poniendo su atributo `type="text/javascript"` podemos introducir un *script* de JS.
- Dentro de etiquetas: para la definición de eventos de un elemento utilizando atributos como `onclick`, que permite asociar el evento *click* con la acción que se especifique.
- En un *script* externo: es la opción más recomendable si el volumen de código a añadir es grande de la misma forma que ocurría con CSS ya que se puede dotar al documento HTML de un comportamiento totalmente diferente sólo cambiando el *script*. La manera de incluir el fichero, con extensión `.js`, es utilizar de nuevo la etiqueta `<style>` fijando su atributo `type="text/javascript"` y añadiendo la ruta al fichero en el atributo `src`.

Cabe destacar una herramienta útil para la programación de *scripts* JS, jQuery, veáse [21]. Se trata de una biblioteca de JS muy extendida para la programación Web que permite un tratamiento simplificado de los elementos del DOM para gestionar las acciones más comunes a la hora de crear la parte dinámica de un sitio Web. Proporciona numerosos y potentes métodos para el tratamiento de eventos, modificación de elementos HTML e integración de la tecnología AJAX (Asynchronous JavaScript And XML,veáse [22]). Añadir jQuery sólo requiere descargar la versión de la librería deseada contenida en un fichero `.js` e incluirla en el documento de la misma forma que se hace con cualquier otro fichero de JS, antes de incluir el fichero en el que se usen las funciones de jQuery.

Las principales ventajas que añade esta librería son: la capacidad de cambiar elementos del DOM sin necesidad de recargar la página y la fácil utilización de la tecnología AJAX con el método homónimo.

La función `jQuery()`, cuyo alias es `$()`, es la encargada de interactuar con los elementos de la página. Aunque tiene más usos, el principal es devolver cada uno de los elementos dentro del documento que concuerden con la expresión que toma como parámetro, llamada selector. Una vez seleccionado el elemento, la librería permite tanto añadir toda clase de eventos que disparen funciones, como una modificación al gusto de ese elemento.

Es una práctica recomendable cuando se trabaja con jQuery incluir todo el código encapsulado en una función llamada `ready()` que garantice un correcto funcionamiento impidiendo la ejecución hasta que el DOM está cargado.

2.2. Soluciones implementadas

A continuación se van a detallar las tres aproximaciones que se estudiaron como posible solución para el desarrollo de la plataforma Web de la aplicación *Posidonia*. La particularidad principal de esta plataforma es la interacción con *Posidonia*, desarrollada en su totalidad en Java [23]. Garantizar que la tecnología elegida permite incorporar el uso de código Java en el servidor es requisito indispensable para que la alternativa sea válida como solución.

A la hora de programar aplicaciones Web hay que tener en cuenta el escenario básico presente según el carácter estático o dinámico de la misma. Una página Web estática consiste en un documento HTML almacenado que no cambia de manera que el servidor se limita a enviar el fichero que contiene dicho documento al navegador para que lo muestre. En el caso de una página dinámica, el contenido de la misma varía según la interacción del usuario y la petición al servidor va acompañada de unos parámetros que condicionan la página devuelta. Ahora, el funcionamiento varía ya que entra en juego la aplicación Web que se encarga de generar el documento HTML a devolver a partir de los parámetros recibidos en la petición. En el lado del cliente este comportamiento es totalmente transparente puesto que recibe el mismo tipo de respuesta en ambos casos. Para llevar a cabo este procesado en el servidor existen distintas alternativas: en este capítulo se hace un recorrido por las características generales de cada una de las posibilidades contempladas, un estudio de su manejo de clases escritas en Java y las ventajas e inconvenientes que presentan.

2.2.1. PHP

PHP (Hypertext Preprocessor), veáse [24], es un lenguaje de código abierto adecuado para el desarrollo Web y que puede ser incrustado en HTML. La característica que distingue a PHP de JavaScript es que el código se ejecuta en el servidor generando el documento HTML y enviándoselo al cliente de manera que éste no puede conocer el código que se ha ejecutado, únicamente los resultados que ha producido. En general está pensado para el procesado de peticiones de usuario en un servidor Web pero se puede utilizar para muchas más cosas aprovechando que es una herramienta con una curva de aprendizaje suave y la vez muy potente para programadores expertos. Hay tres grandes ámbitos en los que puede usarse PHP:

- *Scripts* en el lado del servidor: es el más tradicional y requiere un intérprete PHP, servidor Web y navegador. El navegador se conecta con el servidor Web para obtener los resultados de la ejecución del *script* en PHP.

- *Scripts* en la línea de comandos: en este caso sólo se necesita un intérprete PHP y es especialmente útil para *scripts* que sean ejecutados con frecuencia.
- Escribir aplicaciones de interfaz gráfica: aunque no es la herramienta más idónea, la extensión PHP-GTK (GIMP ToolKit, veáse [25]) permite el desarrollo de dichas aplicaciones.

Otras características adicionales de PHP que lo hacen ser potente son:

- Puede ser utilizado en cualquiera de los sistemas operativos principales del mercado y soporta la mayoría de servidores web de hoy en día, [24].
- En cuanto a la presentación de resultados no se limita a HTML siendo capaz de crear imágenes, archivos PDF (Portable Document Format) o películas FLASH sobre la marcha además de XHTML (eXtensible HyperText Markup Language) o XML (eXtensible Markup Language). Estos archivos pueden ser almacenados en el sistema para ser utilizados como caché en el servidor haciendo posible que el contenido sea dinámico.
- Soporte para bases de datos. Es una de las características más importantes ya que permite la creación de páginas Web con acceso a bases de datos de manera sencilla utilizando una extensión específica (como mySQL, [26]) o una capa de abstracción.
- Cuenta con soporte para comunicarse con otros servicios mediante protocolos como LDAP (Lightweight Directory Access Protocol), IMAP (Internet Message Access Protocol), SNMP (Simple Network Management Protocol), NNTP (Network News Transport Protocol), POP3 (Post Office Protocol) o HTTP (HyperText Transfer Protocol). Además se pueden crear sockets puros para interactuar con otros protocolos.
- Puede crear objetos de Java de forma transparente como objetos de PHP pero sólo de forma experimental en PHP 4 (la última versión es PHP 5.4.8).
- Tiene características de procesamiento de texto e interpretación de documentos en XML.

Un ejemplo sencillo de este lenguaje se ilustra en el Fichero 2.5. Como vemos, el código PHP va entre las etiquetas `<?php` y `?>` y se puede salir y entrar en PHP todas las veces que sea necesario de esta manera. Para que el navegador interprete dicho código como PHP debe ser guardado con la extensión `.php` ya que de lo contrario puede ser tratado como archivo. Esto se puede probar colocando

Fichero 2.5: Ejemplo de PHP

```
<html>
  <head>
  <title>Ejemplo PHP</title>
  </head>
  <body>
    <?php echo '
    <p>Hola Mundo</p>';
    ?>
  </body>
</html>
```

el archivo en `/var/WWW` y ejecutando `http://localhost/fichero.php` en el navegador elegido. Si el funcionamiento es incorrecto intentará descargar el archivo en lugar de mostrar la página en cuyo caso nuestro servidor no tiene instalado o habilitado el intérprete PHP.

El Fichero 2.6 ilustra la utilización de formularios en PHP además de implementar la funcionalidad de envío de correos. Para que este *script* funcione es necesario tener instalado un programa que permita la consulta y envío de correos desde el terminal como el programa `mail` en entornos Linux. Para probar que este código funciona correctamente en `localhost` debemos colocar el archivo `.php` en `/var/WWW` al igual que en el ejemplo anterior y nos aparecerá un formulario a rellenar. En el campo *destino* se deberá incluir una dirección del `localhost` para comprobar después si el correo ha sido enviado o no.

La solución propuesta para interactuar con el paquete `Connections` utilizando PHP es PHP-Java Bridge, veáse [27], una implementación de un protocolo de red basado en XML que puede ser utilizado para conectar *scripts* nativos como los de PHP con una Java Virtual Machine. En general se trata de una serie de librerías que permiten utilizar clases de Java desde *scripts* PHP como si perteneciesen a este último. Aunque hay varios modos de funcionamiento de esta implementación, el que se va a utilizar para los ejemplos de los siguientes apartados es el más sencillo que se basa en colocar las clases ya compiladas en una carpeta de donde el PHP-Java Bridge los importa a través del método *require_once*.

Para probar el funcionamiento del PHP-Java Bridge en el *localhost* nos podemos ayudar de una clase sencilla de Java a cuyos métodos se accede desde un *script* PHP. En este punto es necesario tener instalado un servidor, siendo Tomcat, [28], el recomendado en la documentación de PHP-Java Bridge y cuyas características se detallan en un apartado posterior. Los pasos a seguir son los siguientes:

- Se crea la clase sencilla `ClusterConnections.java` con cuatro métodos que cubren el tratamiento básico de *Strings* que necesitaremos para el proyecto.

Fichero 2.6: Formulario de PHP

```
<?php
  if (!isset($_POST['email'])) {
    ?>
    <form action="<?=$_SERVER['PHP_SELF']?>" method="post">
    <label>
    Nombre:
    <input name="nombre" type="text" />
    </label>
    <label>
    Telefono:
    <input name="telefono" type="text" />
    </label>
    <label>
    Destino:
    <input name="destino" type="text" />
    </label>
    <label>
    Email:
    <input name="email" type="text" />
    </label>
    <label>
    Mensaje:
    <textarea name="mensaje" rows="6" cols="50"></textarea>
    </label>
    <input type="reset" value="Borrar" />
    <input type="submit" value="Enviar" />
    </form>
    <?php
    }else{
    $mensaje.= "\nNombre: ". $_POST['nombre'];
    $mensaje.= "\nEmail: ".$_POST['email'];
    $mensaje.= "\nTelefono: ". $_POST['telefono'];
    $mensaje.= "\nMensaje: \n".$_POST['mensaje'];
    $destino= $_POST['destino'];
    $remitente = $_POST['email'];
    $asunto = "Mensaje enviado por: ".$_POST['nombre'];
    mail($destino,$asunto,$mensaje,"FROM: $remitente");
    ?>
    <p><strong>Mensaje enviado.</strong></p>
    <?php
  }
?>
```

- Se compila dicha clase y se mete en un `.jar`

```
javac ClusterConnections.java
jar cvf ClusterConnections.jar ClusterConnections.class
```

- Se copia `ClusterConnections.jar` en el directorio

```
/var/lib/tomcat7/webapps/JavaBridge/WEB-INF/lib
```

- Se crea un código PHP, mostrado en 2.7 para acceder a la clase de prueba.

Fichero 2.7: Ejemplo de uso de PHP-Java bridge

```
<HTML>
<HEAD><TITLE>Mi primer codigo JavaBridge</TITLE></HEAD>
<BODY>
  <?php
  require_once("http://localhost:8080/JavaBridge/java/Java.inc");
  $clusterConnections = new Java("Connections.ClusterConnections");
  $str = $clusterConnections->diHola();
  $strvector =$clusterConnections->diHolas();
9 $clusterConnections->setMsj('Cris','tina');
  echo '<p>Hola Mundo</p>';
  echo $str;
  echo $strvector[0];
  echo $clusterConnections->getMsj();
  ?>
</BODY>
</HTML>
```

- Basta colocar el archivo `.php` en la carpeta JavaBridge dentro del servidor, en este caso Tomcat, y en el navegador acceder de la siguiente forma:

```
http://localhost:8080/JavaBridge/pruebaBridge.php
```

- En ocasiones, puede aparecer algún mensaje de error indicando que la clase Java a la que hacemos referencia no existe o que el método *require_once* está produciendo fallos. Esto puede ser debido a la configuración de PHP que estamos utilizando. Para modificar la configuración debemos dirigirnos al archivo `php.ini` (normalmente en `/etc/php5` en entornos Linux) y cambiar el valor del atributo *allow_url_include* a *on*. Después de esto será necesario reiniciar el servidor para ver el *script* en funcionamiento.

En ocasiones tenemos la necesidad de utilizar más de una clase perteneciente a un paquete determinado. En este caso tenemos que hacer alguna modificación con respecto al apartado anterior: ahora es necesario utilizar el comando `jar` de una manera distinta para incluir todo el paquete además de incluir un manifiesto (`manifiesto.mf`) en el que indiquemos cuál es la clase que contiene el método `main`. Nos colocamos en el directorio padre de `Connections` (raíz de nuestra estructura de paquetes para el ejemplo) y se ejecuta:

```
jar cvmf manifiesto.mf ClusterConnections.jar Connections/ClusterConnections
    .class
```

En el *script* PHP sólo tenemos que cambiar la forma de crear el objeto utilizando ahora `Connections.ClusterConnections`. Si se produce alguna modificación en las clases de Java deberíamos recompilar, generar un nuevo archivo `.jar`, copiarlo de nuevo en la carpeta correspondiente del servidor y reiniciar este. Para evitar todos estos pasos cada vez que se produzca un cambio se hace un *script* `.sh` en lenguaje bash, que automatiza el proceso.

```
#!/bin/bash
echo 'hola'
javac Connections/ClusterConnections.java
jar cvmf manifiesto.mf ClusterConnections.jar
Connections/ClusterConnections.class
cp ClusterConnections.jar /var/lib/tomcat7/webapps/JavaBridge/WEB-INF/lib
sudo service tomcat7 restart
./updatedJar.sh
```

Son numerosas las ventajas de PHP y es esto lo que le ha llevado a ser el lenguaje del lado del servidor utilizado en el 81 % de las páginas Web a día de hoy, véase [29]. Las ventajas más importantes son:

- Es un lenguaje libre y abierto y está muy bien documentado.
- Funciona sobre casi cualquier plataforma y garantiza una gran velocidad, estabilidad y versatilidad.
- Es fácil de utilizar en todos los aspectos. Su curva de aprendizaje es muy baja y los entornos de desarrollo son fáciles de configurar.
- Fácil integración con bases de datos ya que soporta muchas de ellas, no solo MySQL, véase [30].
- El despliegue de aplicaciones Web es muy sencillo ya que incluso hay disponibles paquetes autoinstalables que incluyen PHP.
- Cuenta con una comunidad de seguidores muy grande, lo que hace que su núcleo esté muy testeado.

El principal inconveniente de esta tecnología como solución para el desarrollo de la interfaz Web de *Posidonia* es que la comunicación con Java no es nativa y depende de un proyecto de terceros. Además presenta ciertos problemas de seguridad:

- Variables globales: en PHP hay una opción que permite la utilización de este tipo de variables, configurando en `php.ini` la opción *register_globals*. De esta forma cualquier usuario puede modificar alguna de las variables al acceder a la URL causando brechas en la seguridad según cómo se hayan utilizado las variables. Para este problema hay dos soluciones posibles: desactivar la opción *register_globals* en `php.ini` y utilizar sólo las variables fijadas en el código.

- Mensajes de error: aún siendo útiles para que el programador detecte los fallos de su código, sirve a los posibles atacantes para descubrir información de la estructura del servidor o de la base de datos. Para evitarlo se pueden usar `.htaccess` o `php.ini` fijando el valor de *error_reporting* a '0'.

- SQL: un usuario puede utilizar un fallo para atacar directamente al servidor de bases de datos. Esto se puede hacer introduciendo como datos de consulta a la base caracteres que modifiquen la consulta a propósito, como '#', que indica que lo siguiente son comentarios. Para evitar estos errores es una buena práctica pasar los datos de entrada por una función propia que elimine este tipo de caracteres especiales en lugar de usar el dato proporcionado por el usuario en la consulta.

- Cabeceras HTTP: pueden ser falsificadas. No se debe confiar en los datos enviados por el usuario en las peticiones HTTP.

- Manipulación de archivos: modificación en la URL que puede permitir al usuario acceder a archivos en el servidor e incluso hacer funcionar *scripts* PHP escritos en él. La solución en este caso sería fijar en `php.ini` el valor de *open_basedir* y *allow_url_fopen* a *off*. Después habría que comprobar el archivo solicitado en una lista de archivos permitidos limitando la salida.

2.2.2. Java Servlets y JSP

Los servlets y JSPs (Java Server Page, [31]) son tecnologías que permiten a los desarrolladores escribir aplicaciones Web de Java que puedan ser ejecutadas en el servidor. Para ello se requiere un servidor Web compatible con los motores servlet llamados *servlet engine* o *container*. Para que funcione correctamente, el motor debe tener acceso al JDK (Java Developement Kit, [32]) que forma parte del JSE (Java Standard Edition, [33]), que a su vez contiene el compilador y las clases principales para trabajar con Java así como el JRE (Java Runtime Enviroment, [34]) necesario para ejecutar las clases ya compiladas. Un servlet es una clase Java que se utiliza para extender las capacidades de servidores con aplicaciones Web. Todos los servlets tienen que implementar el interfaz servlet o

heredar la clase `GenericServlet` o `HttpServlet` en el caso de servicios HTTP específicos. Esto permite definir el ciclo de vida del servlet que se compone de tres fases:

- Instanciación e inicialización: si no hay una instancia del servlet, el contenedor la crea y la inicializa mediante el método `init`.
- Manejo de peticiones: el contenedor crea un hilo por petición que llama al método `service`, encargado de invocar el método apropiado.
- Destrucción: el método `destroy` del servlet permite al contenedor deshacerse de él.

Este manejo de las sucesivas peticiones dota al servlet de la capacidad de realizar el procesado de una Web dinámica, devolviendo el código HTML al navegador utilizando el método `println`, que imprime el documento línea a línea. Esto supone una forma poco eficiente y compleja de codificar un documento HTML, lo que nos lleva a la otra de las tecnologías citadas, JSP, que precisamente soluciona ese problema.

JSP es una tecnología de Java que permite, mediante la utilización de *scripts*, generar contenido dinámico de formato XML y HTML, entre otros. Además de esto se puede utilizar código Java así como una serie de aplicaciones (métodos) de JSP definidos mediante etiquetas. JSP se puede ver como una manera simple de hacer servlets ya que su funcionalidad es la misma: todo lo que se puede hacer utilizando un servlet se puede hacer usando JSP. Así, un servidor de aplicaciones interpreta el código JSP y lo convierte en un servlet que es el encargado de generar el código Web (un documento HTML normalmente que es el que recibe el cliente en su navegador). Una página JSP está constituida en gran parte por HTML, pero contiene fragmentos de código Java incrustado mediante la utilización de etiquetas que están divididas en tres tipos según su funcionalidad dentro de la página:

- *Scripts*: código Java que formará parte del servlet resultante.
- Directivas: son etiquetas a partir de las cuales se genera código utilizado por el motor de JSP para determinar la estructura general del servlet pero no producen código visible al usuario.
- Acciones: etiquetas HTML que se interpretan en la fase de traducción y que determinan el comportamiento del motor de JSPs.

A esto hay que añadir los comentarios JSP, también incrustados en el código mediante las etiquetas `<%--` de inicio y `--%>` de cierre que delimitan las partes de código que han de ser ignoradas. Es importante destacar que hay varias posibilidades para comentar código gracias a los comentarios del lenguaje Java dentro de los *scripts*, los comentarios de JSP y los de HTML. En ocasiones el código es compilado y ejecutado pero no se muestra el resultado en la página lo que puede desembocar en errores en caso de que la parte comentada contenga instrucciones que modifican alguna variable. Hay tres tipos de comentarios posibles:

- Comentario HTML en JSP: utiliza la etiqueta de HTML para los comentarios. El código es compilado y ejecutado pero no es mostrado por el navegador.
- Comentarios JSP: utiliza las etiquetas JSP para comentarios. Todo el código delimitado por estas etiquetas es ignorado por el compilador y no se ejecuta.
- Comentarios Java: estos comentarios dentro de un *scriptlet* son ignorados de la misma forma y no se compilan ni se ejecutan.

Por otra parte, hay tres tipos de elementos *Script*, cada uno de los cuales tiene su propia etiqueta:

- *Scriptlets*: van encerrados entre las etiquetas `<%` de inicio y `%>` de cierre e incluyen una o varias sentencias terminadas por ';'. Dichas sentencias están escritas en lenguaje Java y generalmente describen tareas que requieren más de una instrucción. Se encargan de la generación de cabeceras de respuesta, actualización de bases de datos o ejecución de código que contenga bucles, entre otras cosas.
- Expresiones: llevan las etiquetas `<%=` de inicio y `%>` de cierre, se usan para mostrar texto y son evaluadas en el momento de la llamada. Las expresiones contienen una única sentencia, no terminada en ';', cuyo resultado es convertido a String y mostrado en el lugar en el que esté incrustada la expresión. Si el resultado de la expresión es un tipo primitivo, será automáticamente convertido a un String que represente su valor. Si por el contrario el resultado de la expresión es un objeto, JSP llamará al método `toString` del objeto para obtener un String que represente a dicho objeto. Es importante tener en cuenta que si el objeto resultado no tuviera un método `toString` que devuelva una representación del mismo, se usaría directamente al método `toString` de la clase `Object`, que incluye el nombre de la clase y un código hash para el objeto.

- Declaraciones JSP: usan las etiquetas `<%!` de inicio y `%>` de cierre y sirven para declarar instancias de variables y métodos dentro de un JSP. No producen salida alguna y normalmente se usan junto con expresiones o *scriptlets*.

Las directivas JSP utilizan las etiquetas `<%@` de inicio y `%>` de cierre entre las que se incluyen uno o más atributos con su correspondiente valor. Establece condiciones que van a ser aplicadas a todo el JSP por lo que afectan a la estructura global del servlet resultante. Se utiliza para definir las reglas que seguirá el motor JSP de conversión. Existen tres tipos de directivas:

- *Page*: que controla la estructura del servlet importando clases o configurando el tipo de contenido. De entre los atributos de la directiva destacan *import* y *content type* aunque hay muchos más.
- *Include*: que permite insertar el contenido de otros ficheros en el servlet en el momento de la traducción de JSP a servlet.
- *Taglib*: que extiende la funcionalidad de JSP definiendo etiquetas personalizadas.

Sobre las acciones cabe destacar `jsp: include`, `jsp: forward` y `jsp: param`. Estas acciones permiten añadir contenido a la respuesta o a la petición en el caso de `jsp: param`. Además existen otras acciones que permiten el manejo de JavaBeans. Los JavaBeans, veáse [35] se utilizan para encapsular varios objetos en uno solo cuyos atributos son privados y sólo pueden ser accedidos mediante métodos `get` y `set`. Pueden mandar notificaciones sobre cambios en las propiedades y proporcionan reusabilidad y modularidad a las páginas JSP sustituyendo a grandes cantidades de código embebido.

Además, JSP permite la utilización de clases propias de Java para llevar a cabo la parte de procesado que sea necesario incorporar. Para ello se hace uso del atributo *import* en la directiva *page*. En este atributo debemos incorporar todos los paquetes y clases que queremos tener disponibles dentro de nuestro código JSP. Un ejemplo para el paquete `Connections` de prueba ya utilizado para las pruebas con PHP-Java Bridge sería el que se muestra en el Fichero 2.8.

Fichero 2.8: Ejemplo de JSP

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title> Uso de clases Java</title>
</head>
```

```
<body>
<!-- importar los paquetes y clases necesarias -->
<\%@ page import="Connections.*" \%>
<\%
String arg1 = "Cris";
String arg2 = "tina";
ClusterConnections ejemplo = new ClusterConnections();
String resultado = ejemplo.diHola();
Vector <String> holas = ejemplo.diHolas();
ejemplo.setMsj(arg1,arg2);
String msj = ejemplo.getMsj();
\%>
<p>Informacion obtenida de la clase ClusterConnections</p>
<table cellspacing ="5" cellpadding="5" border="1">
<tr>
<td> diHola:</td>
<td><\%= ejemplo.diHola() \%></td>
</tr>
<tr>
<td> diHolas:</td>
<td><\%= holas(1) \%><\%= holas(2) \%></td>
</tr>
<tr>
<td>getMsj:</td>
<td><\%= msj \%></td>
</tr>
</table>
</body>
</html>
```

En este caso, al ser una alternativa ya basada en Java, la utilización del paquete de pruebas `Connections` resulta más inmediata que en la alternativa anterior. Una vez comprobado que es una solución válida para el desarrollo de la plataforma se continuó haciendo una primera versión de la misma que llegó a tener implementada la funcionalidad más básica de *Posidonia*, junto con las características típicas de una página Web con usuarios registrados.

De entre los posibles servidores adecuados para elegir esta solución: Glassfish (veáse [36], Tomcat (veáse [28]) o JBoss (veáse [37]), se decidió utilizar el primero para este primer prototipo de la Web por sus ventajas respecto a las otras opciones. Glassfish es un servidor de aplicaciones gratuito de código libre desarrollado por Sun Microsystems (ahora Oracle, [38]). Implementa las tecnologías definidas en Java EE (Java Enterprise Edition) de manera que todas las aplicaciones que sigan esta especificación van a poder ser ejecutadas. Además está basado en el código fuente de Sun y Oracle y tiene como base una combinación de un derivado de Apache Tomcat, el servidor Sun Java System Application Server y algún componente adicional para mejorar la escalabilidad y la velocidad. En general presenta una serie de ventajas entre las que se incluyen el hecho de que sea código libre y que cuenta con el respaldo de Oracle. Además es fácil de configurar, cuenta con una interfaz amigable y la documentación sobre administración, uso y desarrollo es amplia. En cuanto a funcionalidad, soporta Java EE y cuenta con integridad total con NetBeans, un entorno de desarrollo integrado libre diseñado

principalmente para trabajar en Java y que también se utilizó en esta versión del proyecto.

Los elementos con los que cuenta la solución implementada con JSP son:

- Dos JSP: uno correspondiente a la página de inicio, `index.jsp` y otro encargado de la zona privada de la Web, `zonaprivada.jsp`. En ellos se define el contenido del documento HTML para ambas páginas así como un pequeño código embebido de JS para dotar de efectos como la aparición de ventanas para iniciar sesión o registrarse. Además se asocian los servlets a las peticiones HTTP que puedan aparecer en la página, en general una por cada formulario para que el usuario introduzca datos.
- Seis servlets: uno para cada una de las funcionalidades básicas que incluye esta solución; desde inicio y cierre de sesión hasta la consulta y envío de trabajos, estas dos últimas propias de *Posidonia*. Cabe destacar que también se introdujo la interacción con una base de datos MySQL para la persistencia de los mismos. Para ello se hizo uso de los paquetes `java.sql` y `java.sql.DriverManager` que nos facilitan la conexión a una base de datos creada en el servidor y la ejecución de consultas de tipo SQL para obtener o guardar en cada momento los datos necesarios.
- Clases auxiliares y paquete `Connections`: se trata de clases de Java necesarias en el códido de los servets para ayudar a que el código sea más claro y un paquete de clases que simulan el comportamiento de *Posidonia* ya que en este punto la conexión real todavía no se llevó a cabo.
- Dos ficheros CSS: uno correspondiente a cada JSP, para definir su apariencia.

Esta distribución de las funciones no es casual y se ajusta a una filosofía llamada MVC, que será explicada con mayor detalle en el capítulo 4 que consiste básicamente en hacer una división entre *Model*, todo lo referente a los datos y su persistencia; *Controller*, encargado de la gestión de peticiones, qué acción invocar y qué vista elegir; y *View*, referente a la apariencia que tiene la aplicación. Una posible implementación de este modelo que se ajusta a las características de las páginas JSP y los Java Servlets es:

1. JavaBeans para la reprentación de los datos (*Model*).
2. Servlets para manejar la petición, rellenar y almacenar los JavaBeans y redirigir la petición al JSP adecuado (*Controller*).

3. JSP para extraer los datos de los JavaBeans y generar la respuesta (*View*).

De esta forma se consigue una separación completa entre los planos de datos, presentación y procesado beneficiándose de las ventajas de cada una de las tecnologías.

Después de utilizar esta alternativa se puede hacer un repaso por las ventajas e inconvenientes que presenta. Puesto que la funcionalidad de las páginas JSP es la misma que la de los servlets, es lógico comenzar por las ventajas de estos últimos puesto que las JSP las extienden.

- Rendimiento: no hay retrasos en las peticiones ya que hay un hilo por petición. Este mejor rendimiento hace que sea la solución elegida para páginas con alta carga, véase [39].
- Menor consumo de memoria: ya que sólo hay una instancia de cada servlet.
- Un servlet mantiene el estado entre distintas invocaciones.
- Las peticiones se ejecutan mediante la invocación de un método gracias a la herencia de la clase `HttpServlet` de Java lo que hace al sistema más eficiente.
- Facilita las tareas típicas de un servidor ya que incluye utilidades para cada una de ellas como autenticación, gestión de errores, etc.
- Proporciona una manera estandarizada de comunicación con el servidor.
- Permite la compartición de datos entre servlets.
- Incorpora todas las ventajas de Java: gran número de APIs, portabilidad entre plataformas y servidores, seguridad, orientación a objetos, gran comunidad de desarrolladores y disponibilidad de código externo.

Las principales ventajas que JSP añade son:

- Ampliamente soportado por plataformas y servidores Web, [31].
- Acceso completo a servlets y tecnologías Java en la parte dinámica.

En cuanto a los inconvenientes, sobre todo con respecto a PHP destacan:

- Menos extendido: frente al 81 % de páginas Web que utilizan PHP, sólo un 2,7 % utilizan Java como lenguaje de programación en el lado del servidor, véase [29].
- Complejidad: la curva de aprendizaje es mayor que la de otras alternativas porque requiere el conocimiento de las especificaciones y de Java.

2.2.3. *Play!* Framework

Por último se presenta la alternativa adoptada como solución para este proyecto. En este apartado se hace un breve repaso por esta solución siguiendo la línea de las dos anteriores pero menos detallado porque en el capítulo siguiente se profundizará en sus características y funcionamiento. Se trata de un *framework*, veáse [40], para desarrolladores Web que reune todas las herramientas necesarias para el despliegue de aplicaciones de la manera más sencilla e integrada. Cuenta con dos versiones de programación, Java y Scala, aunque en este caso el estudio se ha centrado en su versión en Java porque se ajusta a las necesidades de la integración con *Posidonia*. Además facilita la incorporación de todos los elementos adicionales necesarios que ya se han ido mencionando: hojas de estilo, programación del lado del cliente, acceso a bases de datos, etc. No sólo hace sencillo seguir filosofías de programación interesantes como MVC o REST, sino que debido a la arquitectura de aplicaciones que utiliza se fuerza su implementación haciendo que la lectura del código sea más clara.

El grueso de una aplicación desarrollada en *Play!* está dividido en Controladores, programados en Java, Modelos, también programados en Java y Vistas, encargados de generar el HTML resultante con la ayuda de algunas sentencias de Scala utilizando la filosofía MVC vista anteriormente. La comunicación con el paquete `Connections` o cualquier otra librería de Java que queramos utilizar se hace mediante la inclusión en en proyecto del archivo `.jar` en uno de sus directorios destinado para ello, o simplemente incluyendo el código dentro de alguno de los paquetes ya mencionados, ya que serán compilados junto con el resto de código Java presente en el proyecto.

Para entender por qué se ha elegido *Play!* como solución es necesario hacer un recorrido por sus ventajas e inconvenientes después de haber sido utilizado. Las ventajas más importantes son:

- Programación en Java: toda la funcionalidad de la aplicación es programada en Java, de forma que se beneficia de todas sus ventajas y es idóneo para este proyecto porque incorporar el uso del paquete `Connections` es inmediato.

- Productividad del desarrollador: facilita la etapa de desarrollo ya que permite cambiar código en tiempo de ejecución y con sólo refrescar el navegador incorpora los cambios y recompila la aplicación de manera transparente al desarrollador que ve su página con los nuevos cambios a menos que haya errores de compilación que se mostrarían en la pantalla del ordenador, así como en el terminal. La consola de errores mostrada en el navegador no se limita al código Java, sino que incluye el resto de ficheros que deban ser compilados.
- Sin estado y asíncrono: ayuda a crear aplicaciones no bloqueantes más eficientes y modernas con el uso de WebSockets y que siguen la filosofía REST, explicada con detalle en el capítulo 3.
- Es completo: tiene todo lo que se necesita para crear una aplicación y facilita la incorporación de plugins para dar mayor versatilidad.

Los principales inconvenientes son:

- Tecnología joven: no está aún muy extendida porque es relativamente joven pero evoluciona con velocidad.
- Curva de aprendizaje: aunque permite la implementación de cualquier funcionalidad, requiere el aprendizaje de distintos lenguajes lo que supone una barrera de entrada fuerte para pequeñas y medianas aplicaciones; sin embargo, es muy adecuado para una plataforma Web de gran tamaño como potencialmente es *Posidonia*.

Capítulo 3

Implementación de la plataforma Web

3.1. *Play!* framework

Play! framework, [40], es una infraestructura para el desarrollo de aplicaciones Web usando Java y Scala, [41]. Se define como una alternativa ágil al modelo de Java EE enfocada a la productividad de los desarrolladores e incorporando las filosofías de trabajo REST y MVC. Entre sus características principales destacan que es sin estado, escalable y con un consumo muy bajo de recursos: CPU, memoria e hilos.

Cuenta con dos versiones de programación: para desarrolladores de Scala y para desarrolladores de Java. En el caso que nos ocupa en este proyecto, se ha optado por la versión de programación en Java, de manera que toda la información presente en los siguientes apartados hacen referencia a esta modalidad. No obstante, el funcionamiento básico es igual para ambas versiones, cambiando sólo el lenguaje de programación.

3.1.1. MVC y anatomía de una aplicación en Play

Las aplicaciones de *Play!* siguen la arquitectura MVC ya mencionada en el capítulo 2 cuando se describía la alternativa JSP y Java Servlets. Mientras que en ese caso seguir dicha arquitectura era más una decisión de diseño del desarrollador, la anatomía de una aplicación en *Play!* fuerza esta división. Para entender en qué consiste la arquitectura es necesario explicar cada uno de sus componentes:

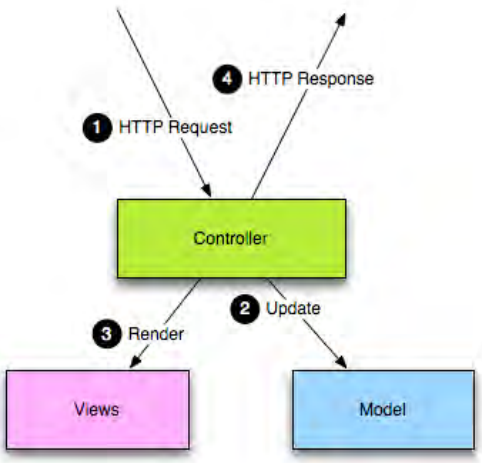


Figura 3.1: Esquema MVC en *Play!* Fuente: [3].

- Modelo: incluye todo lo referente a la información de la aplicación. Normalmente una aplicación utilizará un método adicional para guardar estos datos de manera persistente, como una base de datos, pero el patrón de arquitectura MVC no especifica nada sobre este método de persistencia de forma que todo el tratamiento de la información queda encapsulado bajo el componente Modelo.
- Vista: presenta la información de la parte Modelo para que el usuario pueda interactuar con ella, normalmente a través de una interfaz de usuario, que en el caso de aplicaciones Web es desarrollado a través de un documento HTML.
- Controlador: es la parte encargada de las acciones en la aplicación. Responde a eventos produciendo cambios en la información que contiene el Modelo y generando la Vista. En un modelo de aplicación Web, esto se traduce en recibir peticiones HTTP, aplicar los cambios necesarios atendiendo a esa petición y producir o seleccionar el documento que se manda como respuesta.

El funcionamiento esquematizado de MVC se muestra en la Figura 3.1, donde aparecen de forma ordenada, los pasos que sigue cada petición en una página Web y su interacción con los componentes del paradigma.

En una aplicación de *Play!* estos componentes se organizan dentro de la carpeta *app*, que contiene los ejecutables de Java y Scala. Está dividido en tres carpetas

que se corresponden con las capas de MVC y, opcionalmente, una carpeta llamada *assets* donde se colocan los ficheros de LESS y *CoffeeScript*, de los que se hablará más adelante. Cada una de las carpetas constituye un paquete, aunque se pueden añadir más paquetes personalizados para ajustarse a las necesidades concretas de la aplicación. A continuación se puede ver la estructura completa de una aplicación genérica una vez creada en la consola de *Play!*.

```
/ApplicationName
├── app
│   ├── assets
│   │   ├── stylesheets ..... LESS CSS sources
│   │   └── javascripts ..... CoffeeScript sources
│   ├── controllers ..... Application controllers
│   ├── models ..... Application business layer
│   └── views ..... Templates
├── build.sbt ..... Application build script
├── conf ..... Configurations files
│   ├── application.conf ..... Main configuration file
│   └── routes ..... Routes definition
├── public ..... Public assets
│   ├── stylesheets ..... CSS files
│   ├── javascripts ..... Javascript files
│   └── images ..... Image files
├── project ..... sbt configuration files
│   ├── build.properties ..... Marker for sbt project
│   └── plugins.sbt ..... sbt plugins
├── lib ..... Unmanaged libraries dependencies
├── logs ..... Standard logs folder
│   └── application.log ..... Default log file
├── target ..... Generated stuff
│   └── scala-2.10.0
│       ├── cache
│       ├── classes ..... Compiled class files
│       ├── classes_managed ..... Managed class files
│       ├── resource_managed ..... Managed resources
│       └── src_managed ..... Generated sources
└── test
```

Además del código de la aplicación, la configuración es una parte de vital importancia que va incluída en el directorio *conf*. Este directorio puede incluir opcionalmente ficheros relativos al funcionamiento de la base de datos así como ficheros con datos a introducir en la misma con la extensión `.yml` correspondiente a YAML (YAML Ain't Markup Language, [42]). Los dos archivos más importantes son `application.conf` y `routes`, explicados a continuación.

- `application.conf`: contiene una serie de parámetros estándar de configura-

ción. Al crear una aplicación nueva se carga un fichero con los parámetros por defecto que han de ser modificados acorde a las características de la misma.

- `routes`: archivo de configuración de rutas que se explica con detalle en el apartado siguente.

En *public* se encuentran todos los recursos estáticos de la aplicación divididos, normalmente, en imágenes, ficheros de JS y hojas de estilo CSS. Si se prescinde del uso de los lenguajes *CoffeeScript* y *LESS* integrado en esta plataforma, y se opta por utilizar hojas de estilo y JS tradicionales, deberán ser incluidos aquí.
El directorio *lib* es opcional pero resulta útil para añadir ficheros JAR (Java ARchive, [43]), que son automáticamente añadidos al *classpath*. Resuelve las dependencias de librerías que no están recogidas en el archivo `build.sbt`.
El fichero `build.sbt` es donde se incluyen todas las declaraciones del *build system* y junto con el directorio *project* constituye toda la configuración de plugins y versión de SBT, [44], utilizada en el proyecto.
De los restantes directorios hay que destacar que en *target* es donde se almacenan todos los ficheros generados por la aplicación, como los ficheros resultantes de compilar el contenido de *assets*, es decir, los documentos CSS y JS.

3.1.2. Programación HTTP en Play

Es necesario entender el funcionamiento interno de la arquitectura descrita en el capítulo anterior para situar la función de cada uno de los elementos. La Figura 3.2 muestra el ciclo de una petición HTTP y la forma de interactuar con los bloques de la aplicacion.
En primer lugar, se comienza por buscar en el fichero de rutas, como el mostrado en el Fichero 3.1, la *acción* a la que corresponde la petición y se invoca. Desde el código de la acción se lleva a cabo la actualización de la información guardada en el modelo y, si es necesario, la interacción con la base de datos o mecanismo de persistencia de datos utilizado. Con la información recogida y actualizada se procede a renderizar el fichero del paquete *view* correspondiente que es devuelto al navegador para ser mostrado como respuesta.

Fichero 3.1: Ejemplo del fichero de rutas

```
# Metodo sin parametros
GET   /         controllers.Application.index()

# Parametros extraidos de la ruta
GET   /:name    controllers.Application.greet(name: String)

# Parametros extraidos de la petici n
# i.e. http://myserver.com/?name=Cristina
GET   /         controllers.Application.greet(name)
```

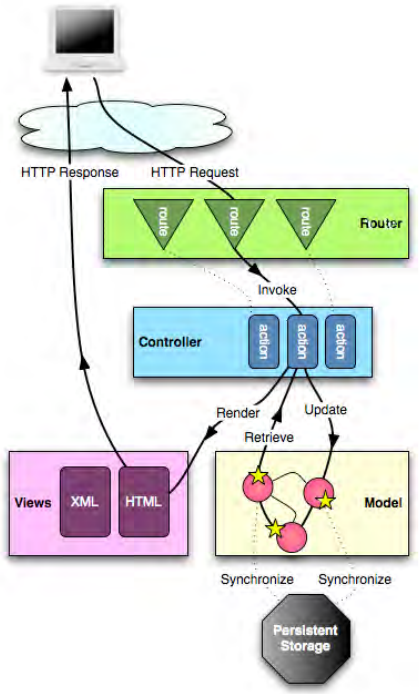


Figura 3.2: Ciclo de vida de una petición HTTP. Fuente: [3].

Esta breve explicación sirve para tener una perspectiva general del funcionamiento de una aplicación. A continuación se va a profundizar en algunos aspectos importantes respecto a la programación HTTP.
En el ciclo descrito, son las llamadas *acciones* las que controlan el grueso de lo que sucede. Una *acción* no es más que un método estático de una de las clases contenidas en el paquete `controllers`. Cada clase de este paquete, a partir de ahora referido como *controlador*, hereda de `play.mvc.Controller` y agrupa varias *acciones*. La función de una *acción*, como ya se puede deducir de la Figura **??**, es procesar los parámetros de una petición y generar un resultado que enviar al cliente.
En el Fichero 3.2 se aprecia que estos métodos pueden tener o no, parámetros de entrada que en caso de existir serán manejados por el *router*, y devuelven un objeto `Result`. Hay distintos tipos de resultado que pueden ser devueltos y están definidos en `play.mvc.Result`. Además, hay numerosos *helpers* contenidos en la clase `play.mvc.Results` que producen resultados estándar de HTTP como los métodos `ok()` de este código de ejemplo.

Fichero 3.2: Ejemplo de un controlador

```
1  package controllers;

   import play.*;
   import play.mvc.*;

   public class Application extends Controller {

       public static Result index() {
           return ok("It works!");
       }
11
       public static Result greet(String name) {
           return ok("Hello " + name);
       }
   }
```

Antes de tener acceso a las *acciones* es preciso traducir la petición HTTP en una llamada a un método de Java. Esta es la función del *router*, que ve la petición como un evento de MVC con dos tipos de información: la ruta y el método HTTP. Las rutas son definidas en el fichero `conf/routes` que se compila, de forma que los errores son mostrados en el navegador. Cada ruta del fichero, correspondiente a una línea, está dividida en tres partes:

- Método HTTP: cualquiera de los métodos aceptados en HTTP.
- Patrón de URI: es la ruta de la petición que puede ser estática o dinámica.
- Llamada a una *acción*: si el método no tiene parámetros consiste únicamente en el nombre completo del método. Si la *acción* tiene parámetros, estos serán buscados en la URI de la petición, bien extraidos de la propia ruta o de los datos de la petición. Cabe mencionar que el orden en la definición de parámetros en la llamada a la *acción* es `<nombre>:<tipo>`, igual que en Scala, como se ve en la segunda ruta del Fichero 3.1.

Varias rutas pueden coincidir con una misma petición, así que la búsqueda se rige por prioridad en el orden de aparición en el fichero, invocandose el método de la primera de las rutas que coincida con la petición.
Este mecanismo de traducción proporcionado por el *router* de *Play!* hace posible el desarrollo de aplicaciones que sigan la filosofía REST (REpresentational State Transfer), conocidas como aplicaciones *RESTful*. Para considerarse *RESTful*, una aplicación debe cumplir los siguientes requisitos:

- La aplicación se divide en recursos.
- Estos recursos están identificados por una URI de forma unívoca.
- Los recursos utilizan los métodos HTTP para tranferir el estado entre el cliente y el recurso.
- El acceso a este recurso debe ser: cliente-servidor, sin estado y dividido en capas.

Otra funcionalidad interesante que proporciona el *router* es el llamado *reverse routing* que consiste en generar una URL dentro de una llamada a Java. Es de gran utilidad para centralizar las URIs en un mismo lugar facilitando hacer cambios, en lugar de escribir rutas en el código. Para cada controlador, se genera un *reverse controller* con los mismos métodos que en lugar de devolver un valor `play.mvc.Result`, devuelve `play.mvc.call` con la URI y el método HTTP.
Además, en relación a la programación HTTP que se lleva a cabo dentro de las *acciones*, hay que destacar tres aspectos:

- Manejar la respuesta, *HTTP Response*: la arquitectura de controladores hace que el tipo de `Result`, el objeto devuelto, se fije automáticamente en función de su valor. Esto quiere decir que se fija la cabecera `Content-type` de la respuesta según corresponda. Sin embargo, se ofrece la posibilidad de modificar cualquier cabecera de la respuesta de forma manual, teniendo siempre en cuenta que cada vez que se establece un valor de cabecera se descartan los valores que se le hayan dado con anterioridad. En el caso de querer servir un fichero local, *Play!* ofrece métodos nativos como los utilizados para los objetos de tipo `Result` que permiten establecer las cabeceras de la respuesta necesarias para devolver un fichero al navegador y que sepa cómo manejarlo, `Content-type` y `Content-disposition`. Por último, en caso de contenido generado dinámicamente cuya extensión es desconocida se puede utilizar una respuesta a trozos, *chunked response*. Se utiliza el mecanismo de transferencia de HTTP llamado `Transfer-Encoding` que sustituye la cabecera `Content-Length` por `Transfer-Encoding` en la respuesta ya que el tamaño del contenido es desconocido. De esta forma el navegador sabe que es una respuesta fraccionada y puede ir mostrándola según el servidor se la envíe.

- Control de sesión: la naturaleza sin estado de esta plataforma hace que la forma de conservar datos entre varias peticiones HTTP sea a través de las *sesiones*. Estos datos no son almacenados en el servidor, sino que se guardan a través de cookies en las siguientes peticiones HTTP por lo que su tamaño está limitado a 4KB y sólo puede almacenar cadenas de caracteres. El control de las *sesiones* desde una acción de un controlador es completo, desde crear o descartar la sesión, guardar datos y consultar los ya almacenados con una serie de métodos que la plataforma proporciona para hacerlo fácilmente. Cabe destacar que esto no está diseñado para ser usado caché ya que para esos propósitos *Play!* cuenta con con otro mecanismo incorporado en la aplicación, veáse [45]. También se ofrece la posibilidad de trabajar con *flash scope*, similar a la *sesión* pero válido únicamente para una petición y menos seguro ya que a diferencia de la anterior se guarda sin firma de forma que el usuario puede modificar su contenido. Únicamente se recomienda su utilización para transportar mensajes de error o éxito en aplicaciones simples que no usen AJAX.

- *Body parser*: en general las peticiones HTTP contienen un cuerpo, al menos en el caso de `POST` y `PUT`, con cualquiera de los formatos especificados por la cabecera `Content-type`. Para poder utilizar su contenido en el código Java de los controladores es necesario hacer una adaptación del formato original a un valor de Java. *Play!* proporciona una colección de *body parsers* por defecto para adaptarse a la mayor parte de los casos de uso: JSON (JavaScript Object Notation, [46]), XML, texto, subida de ficheros, etc. Adicionalmente se puede escribir una implementación propia en Java que se adapte a las necesidades concretas haciendo uso de la interfaz de Java para *body parsers* provista, `play.mvc.Http.RequestBody`.

Otro aspecto importante es la programación asíncrona de HTTP. Internamente *Play!* es completamente asíncrono y gestiona todas las peticiones de manera no bloqueante de forma que conviene programar los *controladores* de la aplicación de la misma manera, sin código que haga retrasarse la ejecución de las *acciones*. Esta configuración por defecto que tienen las aplicaciones puede ser modificada para incluir un mayor número de hilos que permitan una ejecución concurrente de peticiones a través de controladores bloqueantes; sin embargo, mantener el asincronismo en los *controladores* facilita la escalabilidad del sistema y su rápida respuesta bajo condiciones de alta carga.
Cuando el bloqueo es inevitable, *Play!* proporciona dos alternativas para gestionarlo: *promesas* y *actores akka*.
Las promesas no son más que un tipo de respuesta que sustituye al tipo `play.mvc.Result` devuelto normalmente por las *acciones*. Una promesa, `Promise<Result>`, finalmente será sustituida por un objeto `play.mvc.Result`, permitiendo que la acción vuelva de inmediato evitando el bloqueo y sea *Play!* el que sirva la respuesta cuando finalmente la promesa sea reemplazada por el resultado esperado. Esta solución permite que el servidor no esté bloqueado nunca de forma que puede atender peticiones de otros clientes, sin embargo, el cliente Web sí quedaría bloqueado esperando la respuesta. Aunque es una situación mucho menos crítica a nivel de sistema, es posible que empeore las prestaciones de nuestra aplicación según los requisitos que tenga, por lo que la solución que usa *actores akka* puede resultar útil.
La utilización de la librería akka, [47], no se limita a la gestión de código bloqueante, pero utilizado en combinación con los *WebSockets*, explicados a continuación, proporcionan una solución al bloqueo en el servidor y en el cliente. Akka es una herramienta que permite desarrollar aplicaciones concurrentes y distribuidas de manera sencilla. Se basa en el *modelo de actores* para conseguir simplificar tareas al máximo haciendo posible tener sistemas escalables y tolerante a fallos. El modelo de actores consiste en una jerarquía de objetos, *actores*, que contienen estado y comportamiento y que sólo pueden comunicarse a través de mensajes almacenados en un buzón. Para facilitar su comprensión se recomienda pensar en cada uno de estos elementos como personas, cada una de las cuales tiene asignada una tarea: si la tarea no es lo suficientemente sencilla, el actor puede dividirla

y asignar subtareas a otros actores que estarán bajo su supervisión. Este comportamiento hace posible repartir el trabajo de forma concurrente y gestionar los errores de manera eficiente. Esto último se consigue siguiendo la filosofía llamada *let it crash*, que consiste en no afrontar la tolerancia a fallos de una manera defensiva intentando codificar la aplicación para evitar todos los posibles eventos que puedan provocar errores, sino dejando que la aplicación falle, y restaurando sus componentes para poder seguir funcionando. En el caso del modelo de actores, se dota a cada uno de ellos de una estrategia ante los fallos de manera que si un actor detecta un fallo intenta aplicar su estrategia y si no puede solucionarlo pasa el fallo a su supervisor y así sucesivamente. Esto dota a la aplicación de cierta capacidad de repararse a sí misma en caso de fallo sin la necesidad de programar contingencias.
Mediante el uso de actores podemos evitar el bloqueo en el servidor delegando las operaciones bloqueantes a un actor, lo cual permite al *controlador* devolver un resultado de forma inmediata. En este caso, y para evitar el bloqueo en el cliente, entran en juego los *WebSockets*.
Un *WebSocket* es un canal de comunicación bidireccional entre el navegador Web y el servidor. Ambos pueden mandar y recibir mensajes en cualquier momento mientras el canal esté abierto. La mayoría de los navegadores modernos soportan estos canales a través de la interfaz para *WebSockets* de JS. *Play!* proporciona dos mecanismos integrados en la plataforma para utilizar *WebSockets*: usando actores y mediante *callbacks*. El funcionamiento en ambos casos es similar aunque hay algunas diferencias: en caso de abrir un canal utilizando un actor, cada mensaje recibido en el servidor es automáticamente enviado a este que tendrá definido como gestionarlo; si por el contrario abrimos un *WebSocket* mediante *callbacks* se debe definir el método al que llamar cuando se reciben mensajes. En cualquier caso, se requiere una acción que devuelva un objeto `WebSocket` en lugar de `Result` y el correspondiente código JS para hacer la conexión bidireccional.

3.1.3. Plantillas HTML y Vista

Esta sección trata todo lo relativo a la capa *Vista* de las aplicaciones, desde el motor de plantillas de *Play!*, llamado *template engine*, hasta la manera de añadir hojas de estilo que ofrece esta infraestructura. El *template engine* utilizado es Twirl [48], un motor de plantillas basado en Scala cuyas características son:

- Fácil de aprender: se necesita un conocimiento muy básico de Scala y no es necesario aprender un lenguaje nuevo por completo, sino que se utilizan una serie de sentencias de Scala adaptadas a casos de uso generales que permiten generar código HTML de forma sencilla y escalable.

- Es compacto: minimiza el número de caracteres de los ficheros y permite tener un flujo de código comprensible que mejora las habilidades que ya se

tengan para HTML. Además no requiere ninguna herramienta específica, siendo suficiente con cualquier editor de texto.

- Es expresivo y fluido: no es necesario interrumpir partes del código de la aplicación para introducir bloques HTML, sino que todo se concentra en las plantillas `.scala.html`, utilizando una sintaxis limpia y rápida de escribir.

Aunque esta herramienta utiliza Scala, esto no supone un impedimento para los desarrolladores que utilicen la versión Java de la plataforma. Las plantillas no deben contener código pesado y su comunicación con el resto de la aplicación se limita, generalmente, al acceso a métodos de clases del paquete `models` y en este caso se pueden utilizar prácticamente como si fuera Java, con algunos cambios en la sintaxis para adaptarse a Scala. Las plantillas también son compiladas por lo que cualquier error será mostrado en el navegador en tiempo de ejecución facilitando su corrección. La compilación se hace como si fueran funciones estándar de Scala y el resultado es una clase que tiene un método `render()` que puede ser invocado desde las acciones aunque estén programadas en Java.

Fichero 3.3: Ejemplo de plantilla de Play!

```
@(usuario: Usuario, tareas: List[Tarea])

<h1> Bienvenido @usuario.name!</h1>

<ul>
@for(tarea <- tareas) {
  <li>@tarea.getNombre()</li>
}
</ul>
```

En el Fichero 3.3 se muestra un ejemplo sencillo de una plantilla que sirve para hacer un recorrido general por las posibilidades que ofrece este sistema.

- Lo primero que se aprecia es que hay dos tipos de código: parte correspondiente a un documento HTML del que se reconocen varias etiquetas, y parte en Scala. El caracter `@` marca el inicio de una sentencia dinámica, siendo el único caracter especial utilizado en las plantillas que puede escaparse utilizando `@@`. No es necesario marcar el final del bloque, este será deducido del propio código de manera que esta sintaxis sólo acepta sentencias simples como `@usuario.name`. En caso de querer tener sentencias de más de una línea, es necesario encerrarlas entre llaves, como puede verse en el Fichero 3.3 al utilizar el bucle `for`.

- Parámetros: la primera línea en este ejemplo es una sentencia simple que incluye una serie de parámetros para la plantilla, que deben ser declarados al comienzo del fichero ya que son compiladas como funciones. Los parámetros de entrada pueden ser de tipos simples o ser objetos de clases de la aplicación, generalmente pertenecientes al paquete `models` como ya se mencionó.

- Bloques reutilizables: se puede escribir código para ser usado en distintos ficheros, tanto mixto como las plantillas que contienen HTML o puro. Es importante remarcar que tampoco conviene hacer código pesado de esta manera, siempre es mejor poner el código en una clase Java y utilizarlo desde las plantillas mediante la invocación de sus métodos.

El motor de plantillas facilita la generación de documentos HTML que se ajusten en cada momento a las necesidades de la aplicación, pero como ya se explicó en el capítulo 2, se requiere el uso de hojas de estilo, CSS, para definir su apariencia y JS para dar dinamismo e interacción con el cliente. Aunque se pueden utilizar hojas de estilo y ficheros JS tradicionales, *Play!* proporciona un mecanismo integrado para utilizar dos lenguajes que facilitan su programación: *CoffeeScript* y LESS. LESS, [49], es un preprocesador de CSS que permite ampliar su funcionalidad añadiendo mecanismos para enriquecer la creación de hojas de estilo. A diferencia de otros preprocesadores de CSS, permite la compilación en tiempo real y aunque su primera versión utilizaba Ruby, [50], después se cambió por JavaScript. Los mecanismos más importantes que incorpora son:

- Definición de variables: LESS permite la definición de variables con el uso del caracter `'@'` y su asignación utilizando `':'`. Ésto permite centralizar el código y hacer cambios de forma mucho más rápida. Se puede ver un ejemplo en el Fichero 3.4 en las dos primeras líneas.

- *Mixins*: se comporta como una constante o variable que permite incluir todas las propiedades de una clase dentro de otra con sólo incluir su nombre. En el ejemplo se muestra cómo `#header` incorporaría todas las propiedades de `.boxshape`.

- Anidamiento y operadores: similar a cualquier otro lenguaje de programación.

Fichero 3.4: Ejemplo de un fichero LESS

```
1 @color: black;
  @backColor: white;

  .boxshape (@radius: 5px) {
    -webkit-border-radius: @radius;
    -moz-border-radius: @radius;
    border-radius: @radius;
  }

  #header {
11   color: @color;
     background:@backcolor;
     .boxshape
  }

  h2 {
    color: @color;
  }
```

CoffeeScript, [51], es un sencillo lenguaje que compila en JavaScript y proporciona una sintaxis muy simple para escribir código JS. Su propósito es sacar el máximo partido a JavaScript aprovechando todas sus ventajas pero facilitando su uso y comprensión. El código es compilado sin ningún tipo de interpretación y se pueden usar todas las librerías de JS pero reduciendo el tamaño del código y facilitando la expresión de algunas sentencias o estructuras. El Fichero 3.5 muestra el un código en *CoffeeScript* y cómo sería compilado a JS. En el ejemplo se muestra cómo simplifica la utilización de funciones, así como la diferencia con respecto a las variables que en *CoffeeScript* no tienen que ser declaradas, sino que en su primera aparición son declaradas automáticamente. También es destacable que siguiendo el modelo de Ruby, *CoffeeScript* obvia el uso de muchos paréntesis y corchetes sirviéndose de la indentación del código para inferir, en tiempo de compilación, dónde van colocados.

Fichero 3.5: Comparativa de CoffeeScript y JavaScript

```
# -- Coffescript --
square = (x) -> x * x
cube   = (x) -> square(x) * x

/* JavaScript */
var cube, square;

square = function(x) {
  return x * x;
};

cube = function(x) {
  return square(x) * x;
};
```

En *Play!* los recursos compilables han de ser colocados en el directorio `app/assets` como ya se explicó con anterioridad, y son compilados automáticamente cada vez que se refresque el navegador, mostrándose los posibles fallos.

3.1.4. Acceso a una base de datos SQL

Una parte de vital importancia en casi todas las aplicaciones Web es el mecanismo para hacer persistir los datos que se utilizan y es ahí donde entran en juego las bases de datos.
SQL es un lenguaje que nos permite tener acceso y manipular bases de datos relacionales. Una base de datos relacional sigue un modelo en el que se permite establecer interconexiones, llamadas *relaciones*, entre los datos que están guardados en tablas. En la actualidad está consolidado como el paradigma para implementar bases de datos planificadas. Una manera sencilla de entender cómo funciona es hacer una analogía con un conjunto de tablas, como sigue:

- Una tabla es una una *relación*: tiene un nombre y una serie de atributos.
- Cada atributo constituye una cabecera de columna de la tabla.

- El dominio serían los tipos de valores que acepta cada columna.
- Una tupla sería equivalente a cada una de las filas.

Los conceptos más relevantes a la hora de planificar una base de datos de este tipo son los siguientes:

- Superclave y claves candidatas: una clave es un subconjunto de atributos de una relación, y será superclave si este conjunto puede identificar a una tupla de manera única. Cuando una superclave no contiene ningún subconjunto que a su vez constituya una superclave se denomina superclave mínima. Todas las superclaves mínimas de una relación son claves candidatas.
- Clave primaria, *primary key*: la clave primaria es la seleccionada de entre todas las claves candidatas de la relación, y caracteriza a cada tupla dentro de la tabla.
- Clave externa, *foreign key*: una clave perteneciente a una relación llamada *R1* es clave externa de otra relación llamada *R2* si se cumple que es compatible con una de las claves candidatas a ser clave primaria de *R2*. De manera esquemática se entienden las claves externas como referencias de una tabla a otra.
- Restricción de dominio: el valor de cada atributo debe ser atómico del dominio, es decir, no se permiten valores que sean conjuntos.
- Restricción de claves: todas las tuplas de una relación deben ser distintas.
- Restricción de integridad de entidades: ningún valor de la clave primaria puede ser nulo.
- Restricción de integridad referencial: ninguna de las relaciones de un esquema debe tener un valor de una clave externa que no aparezca en alguna tupla de la relación referenciada por la clave. No se puede tener una clave referenciando a un elemento vacío, algo a tener en cuenta cuando se borran tuplas de relaciones con claves externas.
- Restricciones de integridad semántica: son aquellas que limitan el rango de valores de determinados atributos. Para aplicar estas restricciones se utilizan mecanismos de control semántico de los datos.

Este tipo de bases de datos están divididas por lo general en dos planos: la definición de la base de datos y su manipulación. SQL se adapta a esta división contando con dos tipos de lenguaje:

- DDL (Data Definition Language): cuenta con sentencias que permiten crear, alterar o borrar elementos de la base de datos y definir el acceso.

- DML (Data Manipulation Language): las sentencias en este caso son para hacer consultas, insertar, actualizar o borrar instancias en la base de datos.

Está basado en el álgebra relacional, aunque no sigue todos sus teoremas, utiliza el concepto de relación matemática, y se apoya en la teoría de conjuntos y lógica de predicados.
En *Play!*, el acceso a este tipo de base de datos puede llevarse a cabo utilizando tres aproximaciónes: JDBC, JPA y Ebeans.
JDBC (Java DataBase Connectivity) es una API de Java que permite el acceso a bases de datos a través de consultas SQL. Posibilita al desarrollador hacer conexiones a bases de datos y ejecutar operaciones propias de SQL desde lenguaje Java. De las tres alternativas mencionadas para gestionar bases de datos, esta es la de menor nivel de abstracción ya que únicamente proporciona una interfaz para manejar la base de datos desde un código Java; sin embargo, todas las sentencias ejecutadas contra la base de datos son propias de SQL por lo que se requiere un conocimiento en profundidad de este lenguaje. *Play!* proporciona un plugin para gestionar conexiones JDBC que además permite configurar tantas bases de datos como sean necesarias en la aplicación. Para incluir este plugin sólo es necesario añadirlo a las dependencias del proyecto con la sentencia:

```
libraryDependencies += javaJdbc
```

A partir de ese momento, sólo hay que definir las propiedades de la conexión a cada una de las bases de datos en el fichero de configuración del que se habló en el apartado de la arquitectura, `conf/application.conf`, como se muestra en el Fichero 3.6. Por convención, la base de datos por defecto de JDBC debe llamarse `default`, pero se pueden añadir todas las que se precisen. Además, en función de la base de datos utilizada habrá que modificar dichas propiedades de conexión para adaptarse a cada una de ellas, como se puede ver en el último bloque de este ejemplo.

Fichero 3.6: Ejemplo de configuración de JDBC

```
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"

# Orders database
db.orders.driver=org.h2.Driver
db.orders.url="jdbc:h2:mem:orders"

# Customers database
db.customers.driver=org.h2.Driver
db.customers.url="jdbc:h2:mem:customers"

# MySQL engine
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://localhost/playdb"
db.default.user=playdbuser
db.default.password="a strong password"
```

JPA (Java Persistence API) es la segunda de las opciones en orden de abstracción, aunque *Play!* no incorpora una implementación integrada para utilizarlo, basta con añadir la librería de la implementación que se quiera utilizar a las dependencias del proyecto. Si, por ejemplo se utiliza Hibernate, veáse [52], la configuración sería:

```
libraryDependencies ++= Seq(
 javaJpa,
 "org.hibernate" \% "hibernate-entitymanager" \% "3.6.9.Final" //jpa
     implementation
)
```

JPA se define como una interfaz de persistencia desarrollada para la plataforma Java EE que maneja datos relacionales usando Java, beneficiándose de la orientación a objetos propia del lenguaje para interactuar con bases de datos. En JPA se define como entidad una clase de Java cuyo estado se guarda normalmente en una tabla de una base de datos relacional. Cada instancia de esa clase se corresponde con una fila de la tabla. Normalmente estas entidades tienen relaciones con otras entidades que se expresan haciendo uso de anotaciones en la propia clase entidad. En el Fichero 3.7 se ve un par de entidades con las anotaciones que definen a la propia clase como entidad, cuál de los atributos le identifica en la base y la relación con otra de las entidades, `@ManyToOne` y `@OneToMany` en este caso. Otro concepto importante es el lenguaje usado para hacer consultas, JPQL(Java Persistence Query Language, [53]), cuyas sentencias recuerdan a las de SQL pero son ejecutadas sobre la entidad y no sobre las tablas de la base de datos. En el Fichero 3.8 se ve un ejemplo de cómo se ejecutan las consultas, en este caso para obtener una lista con una serie de condiciones definidas en la variable de tipo *String*, que luego constituye la sentencia. Al igual que ocurre con JDBC, la consulta sigue siendo una cadena de caracteres.

Fichero 3.7: Definición de entidades utilizando JPA

```
@Entity
public class User {
    @Id
    private Integer id;
    private String name;

    @OneToMany(mappedBy = "user")
    private List<Task> tasks;
}

@Entity
public class Task {
    @Id
    private Integer id;
    private String description;

    @ManyToOne
    private User user;
}
```

Fichero 3.8: Ejemplo de peticiones usando JPQL

```
public List<Task> getAuthorsByLastName(String lastName) {
    String queryString = "SELECT a FROM Author a " +
                         "WHERE a.lastName IS NULL OR LOWER(a.lastName) = :
                             lastName";
    Query query = getEntityManager().createQuery(queryString);

    query.setParameter("lastName", StringUtils.lowerCase(lastName));
    return query.getResultList();
}
```

Cada llamada JPA debe ir dentro de una transacción, una forma indivisible y atómica de agrupar órdenes que permite aislar los accesos a la base de datos para evitar problemas de concurrencia. Para habilitar llamadas JPA dentro de una *acción* se debe añadir una anotación, `@play.db.jpa.Transactional` que permita que la infraestructura de *Play!* maneje la transacción de forma automática. Si dentro de la acción sólo se realizan consultas, basta con poner el atributo `readOnly` a `true`.

```
  @Transactional(readOnly=true)
2 public static Result index() {
  ...
}
```

La tercera de las alternativas que se presenta es el uso del ORM (Object Relational Mapper) que viene incorporado en *Play!*, Ebean, veáse [54]. Un ORM, como JPA, es una infraestructura que permite convertir datos del modelo relacional a la orientación a objetos. Ebean es una librería que pretende simplicar la implementación de las interfaces para llevar a cabo el mapeo de objetos Java a la base de datos. Difiere de otras implementaciones como JPA en que es sin sesión, lo que significa que elimina gran parte de la problemática que se genera utilizando JPA, como encargarse de cerrar las sesiones, objetos desconectados y obsoletos, etc. Esta es la razón por la que se adapta tan bien a *Play!* que al ser sin estado siguiendo la filosofía REST, no tiene que preocuparse por el manejo del estado de las entidades que caracteriza a JPA. Sin embargo, hay una parte de JPA que Ebean utiliza: las anotaciones para mapear las entidades que se encuentran contenidas en el paquete `javax.persistence` y se utilizan de la misma forma. Además de mapear las clases, es importante que hereden de la clase `Model` de *Play!* que facilita el trabajo añadiendo métodos a la entidad como se muestra en el ejemplo del Fichero 3.9. También se incluye la definición de un campo estático que constituye un *finder* para una entidad de tipo `Task` con un identificador de tipo `Long`. Esta es una herramienta que permite hacer consultas desde fuera de la entidad de manera muy simple, haciendo uso de los métodos en la clase `play.db.ebean.Model.Finder`, veáse [55]. Aunque puede accederse a este campo desde fuera de la entidad, se recomienda encapsular en estas entidades lo relacionado con los datos y que no sean un mero envoltorio vacío con los atributos de la tabla. En el ejemplo se ven tres métodos estaticos que sirven para ilustrar esto y que utilizan distintos métodos para hacer consultas simples y con restricciones en el orden de presentación o los criterios utilizados para filtrar los

datos extraídos.

Fichero 3.9: Ejemplo de una entidad Ebean

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.ebean.*;
import play.data.format.*;
import play.data.validation.*;

@Entity
public class Task extends Model {

  @Id
  @Constraints.Min(10)
  public Long id;

  @Constraints.Required
  public String name;

  public boolean done;

  @Formats.DateTime(pattern="dd/MM/yyyy")
  public Date dueDate = new Date();

  public static Finder<Long,Task> find = new Finder<Long,Task>(
    Long.class, Task.class
  );

  public static List<Task> findTasks(){
   // Find all tasks
   List<Task> tasks = find.all();
   return tasks;
  }

  public static void deleteTaskById(Long id){
      // Delete a task by ID
      find.ref(id).delete();
  }

  public static List<Task> queryExample(){
      return find.where()
      .ilike("name", "%coco%")
      .orderBy("dueDate asc")
      .findPagingList(25)
      .setFetchAhead(false)
      .getPage(1)
      .getList();
  }

}
```

Ebean usa transacciones por defecto y estas son creadas y retiradas antes y después de cada acción sobre la base de datos. Si se quiere hacer más de una acción en la misma transacción se pueden usar las clases `TxRunnable` y `TxCallable` o anotar con `@Transactional` la *acción* como se hacía en JPA. Para una aproximación más tradicional se puede manejar de forma explícita el comienzo y el final de cada transacción. En el Fichero 3.10 aparecen las distintas posibilidades.

Fichero 3.10: Ejemplo de transacciones con Ebean

```
// Usando TxRunnable
Ebean.execute(new TxRunnable() {
  public void run() {

    // Primera accion
    User user = Ebean.find(User.class, 1);
    ...
    // Siguientes acciones
    Ebean.save(user);
    Ebean.delete(order);
    ...
  }
});

//Aproximación tradicional
 Ebean.beginTransaction();
try {
    // fetch some stuff...
    User u = Ebean.find(User.class, 1);
    ...

    // save or delete stuff...
    Ebean.save(u);
    ...

    Ebean.commitTransaction();

} finally {
    Ebean.endTransaction();
}
```

La configuración de Ebean consiste en habilitarlo añadiendo el paquete `javaEbean` a las dependencias del proyecto, de la misma forma que se hizo en las dos alternativas anteriores:
`libraryDependencies += javaEbean`
Además, hay que definir al menos un servidor Ebean por defecto, aunque se pueden configurar todos los que sean necesarios y definir las clases que mapean cada uno o la base de datos que utiliza. Para hacerlo se añaden al fichero de configuración `conf/application.conf` sentencias como esta:
`ebean.default="models.*"`
En este caso define el servidor por defecto que puede mapear todas las entidades dentro del paquete `models`.

3.2. Requisitos y funcionalidad de *Posidonia*

Tal y como se explicó en [5], la estructura por capas de *Posidonia* supone que la introducción de una interfaz Web no debería introducir demasiados cambios en la capa inferior llamada `Connections` en la Figura 1.6 aunque hay que hacer algunos cambios en el código para acomodarse a la filosofía de la plataforma Web introducida en este proyecto. Si se hace una explicación de los cambios introducidos por capas:

- `SSHCondor`: no requiere cambios adicionales por el hecho de introducir una interfaz Web, ya que la infraestructura de simulación no ha sufrido cambios salvo el cambio de *scheduler*, lo que es ajeno a la interfaz Web.

- `Connections`: en esta capa hay que introducir ciertos cambios para introducir las nuevas funcionalidades presentadas. De esta forma, hay cambios sensibles en esta capa haciendo posible la simultaneidad de conexiones a diferentes *hosts* y la compartición de trabajos, algo que se controla desde la parte de controladores de la interfaz Web pero que utiliza los métodos de esta capa. En concreto, se añaden nuevos métodos que implementan estas funcionalidades de forma que las otras versiones de *Posidonia* utilicen el mismo código que la plataforma Web. Sin embargo, el principal cambio se encuentra en el servidor TCP, cuya inteligencia ha aumentado notablemente. De esta forma, se aumentan los mensajes que se envían desde el cluster, tal y como se explica en la Sección 3.3.2, y las acciones que se toman desde el servidor TCP también se modifican, teniendo que utilizar los métodos proporcionados por los controladores de la interfaz Web.

- `Interfaz gráfica`: no sufre cambios, en su versión de escritorio y Android, debido a la introducción de la interfaz Web haciendo uso de la estructura por capas definida en [5].

Por otro lado, es necesaria la introducción de un nuevo fichero en el *cluster*, llamado `WebConfig`, en el que se introduce información que necesita la parte de `Connections` del *cluster* para comunicarse con el servidor Web en el que se encuentra alojada la plataforma. En este fichero hay dos campos: `WebServerPort`, donde se indica en qué puerto está activo el servidor TCP levantado en el servidor Web que se comunica con el *cluster*, y `WebServerHost`, donde aparece la dirección IP en la que se encuentra el servidor Web. Este fichero se modifica de forma automática al lanzarse el servidor Web, de forma que la información que está contenida en este fichero siempre está actualizada.

3.3. Arquitectura de la aplicación e implementación

En esta sección se hace un estudio detallado de la arquitectura de la aplicación Web desarrollada y su implementación siguiendo la división propuesta por el esquema MVC para facilitar su comprensión. Además se ha añadido un cuarto apartado en el que se describe la programación del lado del cliente que no corresponde a ninguna de las otras divisiones pero es una parte fundamental del sistema. A lo largo de cada apartado se explica la evolución desde el diseño hasta la implementación incluyendo las tecnologías usadas y cómo se han integrado en la plataforma.

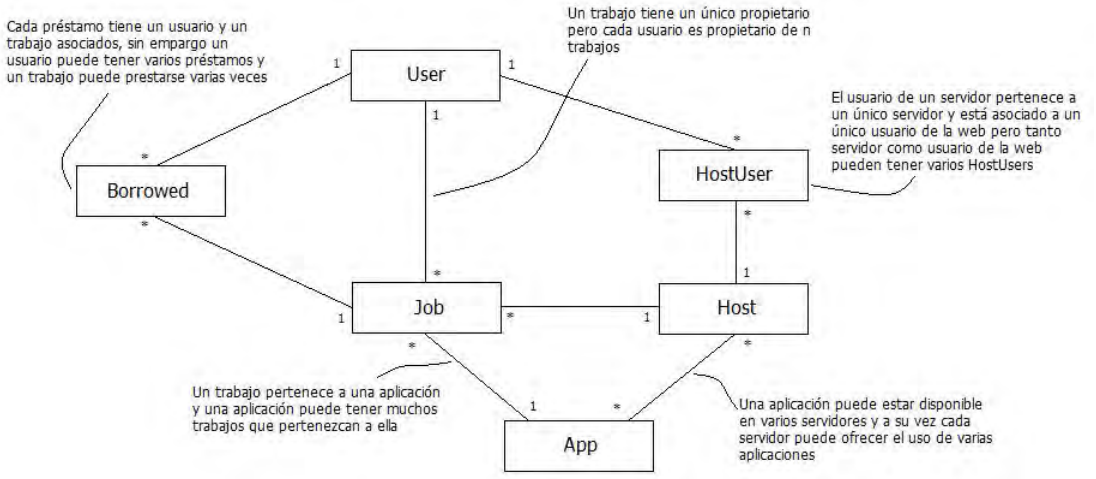


Figura 3.3: Modelo de datos representando la plataforma

Antes de comenzar conviene hacer una separación entre dos planos a nivel conceptual: uno nativo de la infraestructura Web de la aplicación que engloba toda la gestión de la base de datos, usuarios de la Web y funcionalidades propias y otro que es heredado de la aplicación original, *Posidonia* y que implementa toda su funcionalidad sirviéndose del API creado para esa aplicación detallado en el Capítulo 1.

3.3.1. Modelo y base de datos

El primer bloque de diseño que se abordó en este proyecto fue la creación de la base de datos, es decir, el *modelo* de la aplicación ya que es la parte central sobre la que se basa gran parte del desarrollo. Hay que prestar especial atención a su planteamiento ya que constituye la representación de los objetos y conceptos manejados por la aplicación. Una estrategia a seguir para llevar a cabo esta tarea es hacer una representación en código de una situación real como se explica en [56]. Una forma conveniente de abordar esta estrategia es tomar un caso de uso de la aplicación, escribir todos los sustantivos y dibujar un esquema con las relaciones entre ellos como el mostrado en la Figura 3.3. Un usuario de la Web, *User*, puede ser propietario de n trabajos *Job* alojados cada uno de ellos en un servidor específico, *Host*, y perteneciente a una aplicación concreta, *App*, de las disponibles en ese servidor. Como cada servidor requiere su propia identificación (par usuario-contraseña) y un usuario de la Web puede tener acceso a varios servidores, es necesaria una figura intermedia, el *HostUser*, que representa la identidad en cada uno de los servidores disponibles. Por último, un usuario de la Web puede tener acceso a trabajos que otros usuarios le hayan compartido, *Borrowed*, que asocia un trabajo con un usuario que tiene acceso a él sin ser su propietario.

Los lenguajes orientados a objetos son muy apropiados para representar un modelo de datos del tipo mostrado en la Figura 3.3 ya que agrupan un conjunto de datos y su comportamiento en una única entidad. De las tres alternativas que *Play!* ofrece para manejar el acceso y manipulación de las bases de datos, se ha elegido Ebeans para este proyecto porque se adapta muy bien a la estructura del modelo de datos diseñado y es muy sencillo de implementar y mantener. Partiendo del esquema realizado, el siguiente paso es crear las clases que constituyan las entidades para la base de datos. En este paso es necesario definir los atributos de cada una de estas entidades y añadir los métodos que se encargan de todo lo referente a las consultas a la base de datos para ser utilizados desde el resto de la aplicación. Se crea una clase por cada uno de los elementos mencionados en el modelo de datos, dando como resultado el esquema de entidades completo de la base de datos para este sistema representado en la Figura 3.4.
A continuación se explica con detalle cada una de las entidades que forman parte de este modelo, que aparecen en éste árbol del directorio `models`:

```
/models
├── ActorsTCP.java
├── App.java
├── Borrowed.java
├── Host.java
├── HostUser.java
├── InfoJob.java
├── Job.java
└── User.java
```

Como estas clases serán mapeadas a tablas en una base de datos relacional tienen que cumplir una condición que es tener una clave primaria, es decir, uno o un conjunto de sus atributos debe indentificar de manera unívoca cada una de las instancias de la clase dentro de la tabla. Aunque puede elegirse una clave primaria natural, es decir, uno de los atributos propios de la clase que cumpla esta condición, cuando se utiliza un ORM, se recomienda el uso de una clave primaria sintética o, lo que es lo mismo, añadir un atributo extra de tipo numérico y marcarlo como identificador de la entidad con la anotación `@Id`. Cada una de las clases utilizadas en este modelo utilizan claves primarias sintéticas que van a ser obviadas en la explicación ya que carecen de significado en el modelo.

- `User`: corresponde a cada uno de los usuarios de la aplicación Web. Para hacer uso de la aplicación, un usuario debe estar registrado, y tener un identificador de usuario único en la Web, que en el caso de esta aplicación es una dirección de correo electrónico. Requiere, además, otros atributos simples: contraseña de acceso a la Web, un nombre que no tiene que ser único y un flag que identifica al usuario como administrador. El resto de atributos son relaciones con otras entidades, a saber: una lista de trabajos, `Job`, de los que es propietario; una lista de sus identidades en todos los servidores

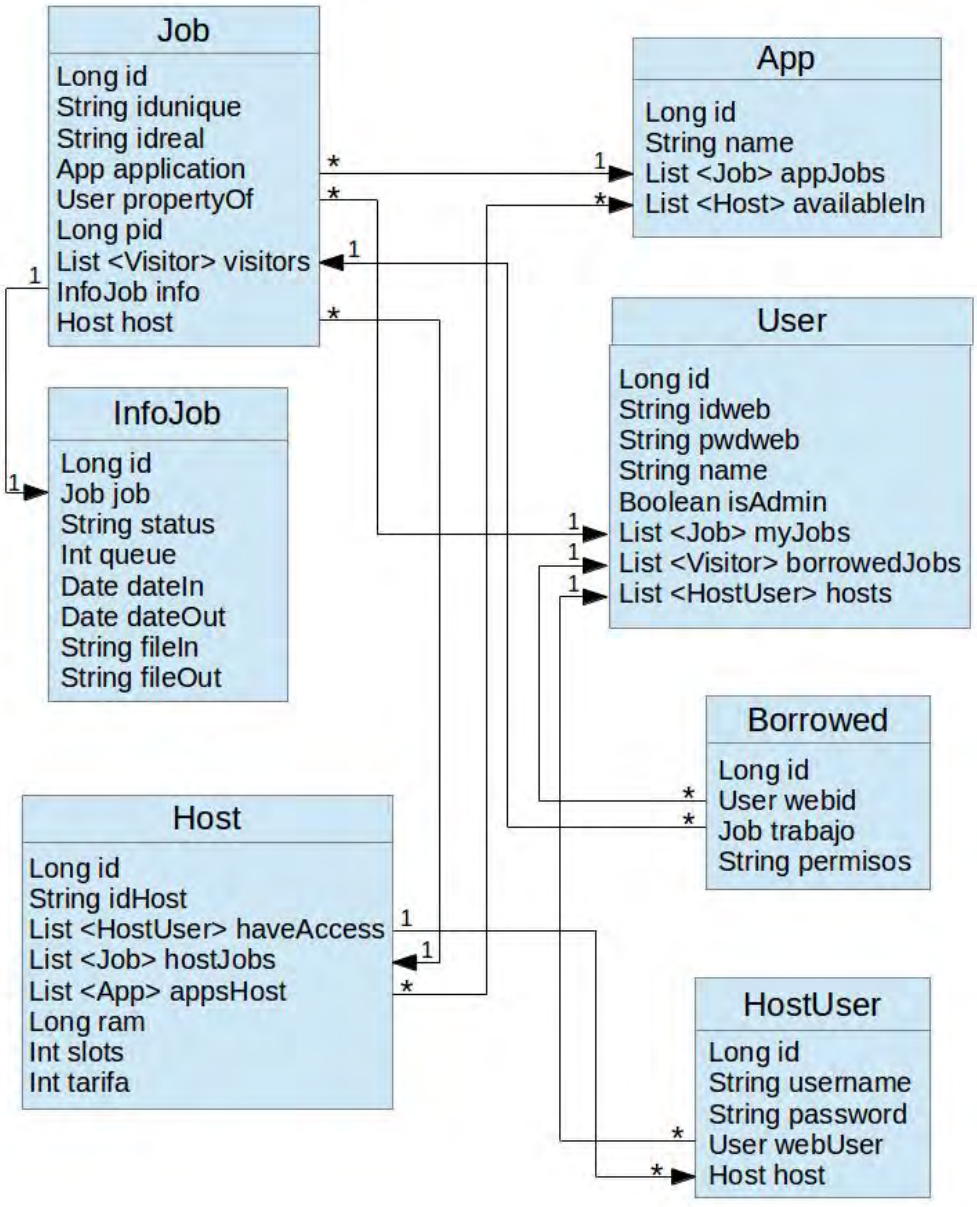


Figura 3.4: Esquema de la base de datos

a los que tiene acceso, `HostUser`; y una lista con todos los trabajos que le han sido compartidos, `Borrowed`.

- `Job`: los trabajos son el elemento central de esta aplicación, donde más cantidad de información se concentra y son representados en esta entidad. Poner en esta clase todos los atributos simples con los que cuenta esta entidad dificultaría su comprensión: por eso es recomendable, en estos casos, agrupar los atributos simples y escribir una nueva clase con una relación de uno a uno con la original. Tras esto, los atributos simples que se han conservado en la entidad `Job` son: el identificador único del trabajo heredado de la aplicación original, *Posidonia*, el nombre que el usuario le da para enviarlo, y el PID (Process IDentifier) del trabajo en el servidor. Las relaciones de esta entidad son: la aplicación a la que pertenece el trabajo, `App`, su propietario de tipo `User`, el servidor en el que se ejecuta, `Host`, y una lista con todas las veces que se ha compartido el trabajo, `Borrowed`. Adicionalmente tiene una relación uno a uno ya mencionada con la entidad `InfoJob` que contiene más datos sobre el trabajo.

- `InfoJob`: esta entidad tiene atributos simples relativos al trabajo al que pertenecen representado por una relación con la entidad `Job`. Los atributos representan: el estado del trabajo en el servidor, su fecha de entrada y salida del servidor y la ruta en la que están guardados los archivos de entrada y salida en caso de que el trabajo esté compartido.

- `App`: esta entidad representa las posibles aplicaciones que pueden estar disponibles en los servidores y a las que pueden pertenecer los trabajos: no es más que el nombre del programa de simulación en el que se quieren ejecutar los trabajos, como por ejemplo, Matlab, [57]. Contiene el nombre de la aplicación y dos relaciones: una lista con los trabajos, `Job`, que pertenecen a esa aplicación, y otra lista con los servidores `Host` en los que esa aplicación está disponible.

- `Host`: cada servidor está definido por su dirección IP y el nombre por el que se le identifica. Además cuenta con dos campos con fines estadísticos que representan la memoria RAM (Random-Access Memory) y el número de slots disponibles en el mismo. En cuanto a relaciones con otras entidades tiene: una lista de usuarios del servidor, `HostUser`, una lista de trabajos ejecutados en él, `Job`, y una lista de todas las aplicaciones que hay disponibles, `App`.

- `HostUser`: esta entidad entra en juego para permitir la conexión a más de un servidor de cada uno de los usuarios de la plataforma Web. Cada servidor requiere su propio par usuario-contraseña independiente de los demás, por lo que se hace necesaria una clase que contenga la relación entre el servidor

al que pertenece, `Host`, ese par usuario-contraseña y el usuario de la Web al que corresponde, `User`.

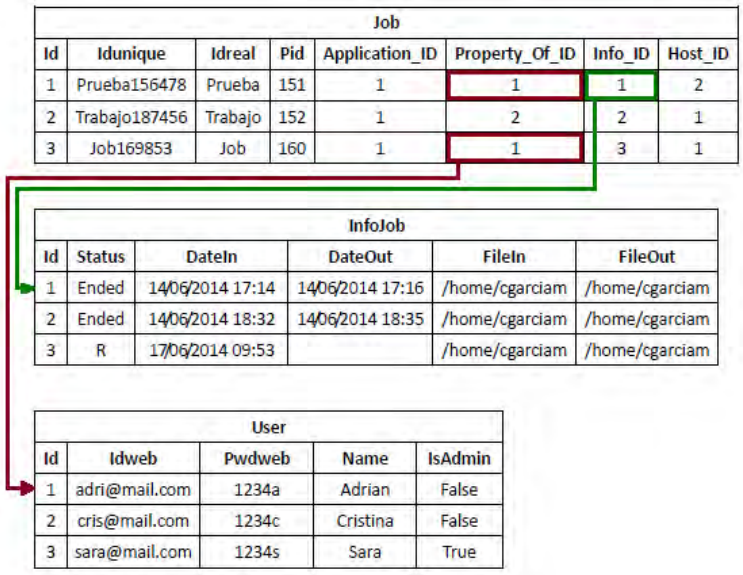
- `Borrowed`: cada una de las instancias de esta entidad representan un préstamo de un trabajo de la aplicación. Para añadir la funcionalidad de compartición de trabajos entre los usuarios de la Web se necesita poder identificar el préstamo con el trabajo objeto de compartición, `Job`, y el usuario al que se permite su uso, `User`. Cabe destacar que no es necesario hacer una relación directa en este punto con el propietario del trabajo, autor del préstamo, ya que va incluido en la entidad del trabajo.

Una vez descrito el entramado de entidades hay que anotar las relaciones para que el funcionamiento de las consultas e instrucciones contra la base de datos sea el esperado. Hay tres tipos de relación que podemos ver en la Figura 3.4: *OneToMany*, *ManyToMany*, y *OneToOne*.

- *OneToMany* - *ManyToOne*: es una relación de uno a muchos, representada esquemáticamente por un asterisco en uno de los extremos de la relación y un '1' en el otro. En este tipo de relaciones es necesario definir cual de los dos extremos es el dominante, determinado en la Figura 3.4 por el inicio de la flecha. En el caso de las relaciones de uno a muchos está claro cuál es el lado dominante, el lado que contiene una referencia a la otra entidad, que ha de ser anotado con `@ManyToOne`. Aunque en estos casos poner una anotación `@OneToMany` en el otro extremo de la relación no añade información a la base de datos, se recomienda definir las relaciones en ambas entidades para garantizar que es bidireccional y ninguna de las partes es inconsistente. Cuando se añade la anotación `@OneToMany` hay que indicar qué propiedad de la otra entidad contiene la relación que se está mapeando mediante el atributo `mappedBy`. En la Figura 3.5 se muestra la representación de este tipo de relación en la base de datos entre las entidades `Job` y `User`, donde la primera es la parte dominante y, por tanto, la que guarda una única referencia hacia la otra parte.

- *OneToOne*: es una relación de uno a uno; en el caso que se presenta, se da únicamente entre las clases `InfoJob` y `Job`: un trabajo tiene una información del trabajo y cada información pertenece a un único trabajo. A nivel de la base de datos no hay diferencia entre esta relación y la anterior, como puede verse en la Figura 3.5, sin embargo la parte dominante en este caso tiene que ser decidida por el diseñador ya que cualquiera de los dos extremos puede serlo. Dado que la clase principal en este caso es `Job` y la otra sólo la extiende, tiene sentido que sea la dominante y sea en `InfoJob` donde se utilice el atributo `mappedBy` para completar la anotación `@OneToOne`.

- *ManyToMany*: esta es la más compleja de las relaciones y difiere de las anteriores. Como ninguno de los lados de la relación tiene una referencia única, no se puede guardar en la base de datos en ninguna de las



Job

Id	Idunique	Idreal	Pid	Application_ID	Property_Of_ID	Info_ID	Host_ID
1	Prueba156478	Prueba	151	1	1	1	2
2	Trabajo187456	Trabajo	152	1	2	2	1
3	Job169853	Job	160	1	1	3	1

InfoJob

Id	Status	DateIn	DateOut	FileIn	FileOut
1	Ended	14/06/2014 17:14	14/06/2014 17:16	/home/cgarciam	/home/cgarciam
2	Ended	14/06/2014 18:32	14/06/2014 18:35	/home/cgarciam	/home/cgarciam
3	R	17/06/2014 09:53		/home/cgarciam	/home/cgarciam

User

Id	Idweb	Pwdweb	Name	IsAdmin
1	adri@mail.com	1234a	Adrian	False
2	cris@mail.com	1234c	Cristina	False
3	sara@mail.com	1234s	Sara	True

Figura 3.5: *One-to-Many* y *One-to-One* en la base de datos

dos tablas ya que se rompería la restricción de dominio explicada en el apartado 3.1.4 y que impide guargar datos múltiples en las tablas. La solución es crear una tabla en la que se plasman las relaciones entre los identificadores de ambas clases como se ve en la Figura 3.6 entre las clases `Host` y `App`. En este caso también queda en manos del diseñador decidir qué parte es la dominante y cuál deberá hacer uso del atributo `mappedBy` en su anotación `@ManyToMany`. Además, estas relaciones tiene que guardarse y eliminarse de manera explícita mediante los métodos `saveManyToManyAssociations()` y `deleteManyToManyAssociations()` pertenecientes a `play.db.ebean.Model`.

Estas son las clases que componen el modelo de datos de la aplicación diseñada; sin embargo existe una clase más en el paquete `models` llamada `ActorsTCP.java` que contiene el código del sistema de actores akka que se ha utilizado. Por su utilización de la información en la base de datos se decidió incorporarla en este paquete; sin embargo, para explicar todo el sistema de actores con detalle es conveniente explicar su contenido en el siguiente apartado, relativo a los *controladores*.

3.3.2. Controlador e incorporación del paquete *Connections*

Como ya se comentó en la descripción de *Play!*, los controladores suponen la parte activa de la aplicación que se encarga de reaccionar a los eventos de la misma en

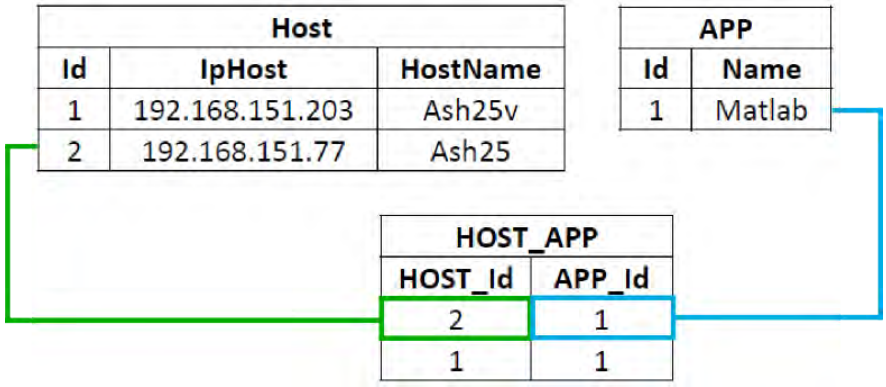


Figura 3.6: *Many-to-Many* en la base de datos

el servidor, es decir, manejar las peticiones y generar una respuesta. El paquete `controllers` de la aplicación contiene varias clases cada una de las cuales engloba *acciones* relativas a un mismo grupo de funcionalidades, permitiendo organizar el código fácilmente. Su arquitectura sigue este esquema:

```
/controllers
├── Admin.java
├── AdminSecured.java
├── Application.java
├── Jobs.java
└── Secured.java
```

- `Application.java`: es el controlador principal de la aplicación y contiene todas las *acciones* que son invocadas desde la página de inicio de la Web o para volver a ella. También controla el *router* utilizado en la parte del cliente, JavaScript, y el *WebSocket* que se utiliza para gestionar los mensajes TCP (Transmission Control Protocol). Las acciones de las que se ocupa este controlador incluyen:

  - *Log-in*: la entrada a la zona privada se hace a través de la identificación del usuario en la Web. Para ello se envían en la petición HTTP el par usuario-contraseña recogidos en el formulario y se comprueban contra la base de datos como parte de la acción `authenticate()`. Si la comprobación da un resultado positivo se borran los datos de la sesión HTTP que pueda haber y se crea una nueva sesión con cuatro campos:

    1. *Idweb*: el identificador único del usuario que accede a la zona privada.
    2. Ventana: en la zona privada hay cuatro contenidos diferentes que pueden ser mostrados.

       a) Historial

b) Envío de trabajos

c) Mi cuenta

d) Instrucciones

Para recordar de una petición a la siguiente en qué ventana estaba el usuario se guarda una cadena de caracteres que la identifique en este campo.

3. Orden: permite recordar entre peticiones de la misma sesión con qué criterio está ordenada la tabla en la ventana del historial.
4. `IsAdmin`: resulta útil guardar una referencia para saber si el usuario al que pertenece la sesión es administrador o usuario normal. Este campo contiene el valor `true` si el usuario es administrador y `false` en caso contrario.

Después se redirige a la acción que permite el acceso a la zona privada, `accessToPrivate()`, que se corresponde en el fichero de rutas con la petición `GET /privateArea`, y que únicamente devuelve la plantilla correspondiente a la zona privada renderizada. Si, por el contrario, el par usuario-contraseña no coincide con ninguno de los guardados en la base de datos se redirecciona la acción correspondiente a la ruta `GET /login`, y que devuelve la plantilla renderizada de la página que contiene el formulario de inicio de sesión mostrando un mensaje de error que indique por qué no ha sido realizado.

- *Log-out*: como en todas las aplicaciones Web con acceso para usuarios se incluye una acción que permite borrar los datos de la sesión y abandonar la zona privada, devolviendo al navegador la plantilla de la página de inicio nuevamente.
- Registro de usuarios: para obtener el acceso a la aplicación, se proporciona una opción de registro en la misma a través de un formulario que pide los datos mínimos para rellenar la entidad `User`. Cuando se introduce una dirección de correo válida, es decir, que no corresponde a ningún otro usuario, una contraseña, y el nombre con el que el usuario quiere aparecer en la Web, se crea una nueva instancia en la base de datos y se le devuelve el acceso a la zona privada. Al ser un usuario nuevo, no tendrá aún acceso a ningún servidor puesto que no ha introducido sus credenciales para ello.
- Cambiar la contraseña de usuario: en la zona privada se ofrece la posibilidad al usuario de modificar su contraseña a través de un formulario estándar para estos casos con tres campos, la contraseña actual y la nueva dos veces para asegurar que no ha cometido errores. Para ello, la acción `changePwd()` hace la actualización pertinente en la base de datos tras comprobar que la contraseña antigua es correcta y las dos contraseñas nuevas coinciden.

- *WebSocket*: aunque la comunicación mediante sockets será explicada más adelante, aquí podemos adelantar que la parte de la conexión del servidor es gestionada mediante la acción `msgTCPStream()`, que se encarga de abrir el socket y devolverlo como resultado de la acción y cuyo código se muestra en el Fichero 3.11.

Fichero 3.11: Código del WebSocket

```
public static WebSocket<JsonNode> msgTCPStream(final Long timestamp){
    return new WebSocket<JsonNode>(){
        public void onReady(WebSocket.In<JsonNode> in, final WebSocket.
            Out<JsonNode> out){
            in.onMessage(new Callback<JsonNode>()) {
            //Código a ejecutar cuando recibe un mensaje en el servidor
            }
            //Código para crear el actor Listener asociado al socket

        }
    };
}
```

- jsRoutes: para poder hacer llamadas AJAX desde el código JavaScript, es necesario incorporar un *router* que permita identificar las acciones sin necesidad de dar explícitamente la URL correspondiente. Para definirlo se incluye en el controlador una acción, en este caso llamada `javascriptRoutes()`, que fija el tipo de la respuesta como `text/javascript` y devuelve el resultado de asociar todas las acciones que se incluyan a un identificador que podrá ser utilizado desde cualquier fichero de JavaScript siempre que la acción esté correctamente declarada en el fichero de rutas y en el documento HTML que se esté usando:

```
GET /assets/javascripts/routes controllers.Application.
    javascriptRoutes()

<script
      type="text/javascript"
      src="@routes.Application.javascriptRoutes()">
</script>
```

- `Jobs.java`: este controlador maneja todo lo relativo a los trabajos en la aplicación. Se trata del nexo de unión principal de la aplicación Web con el paquete `Connections` de *Posidonia*. Las tareas que realiza son:

  - Enviar los trabajos al cluster: se encarga de ello la acción `enviarAlCluster()` que recoge los datos del trabajo: nombre, número de slots, archivos de entrada, nombre del script a ejecutar, aplicación a la que pertenece y servidor al que se envía. Con todo ello hace la conexión necesaria con el servidor seleccionado, pone el identificador único al trabajo y guarda en la base de datos una instancia de la entidad `Job` con los datos recogidos en la petición a la espera de recibir, vía TCP,

el resto de información proveniente del cluster. El identificador único de los trabajos consiste en la concatenación del nombre que el usuario da a su trabajo cuando lo envía y la hora de envío en milisegundos.

- Eliminar trabajos: el paquete `Connections` proporciona dos mecanismos para eliminar trabajos, en función de su estado (finalizado o no finalizado). La aplicación centraliza la tarea de eliminar trabajos en una única acción, `deleteJob()` que selecciona en función de los trabajos a eliminar cuál de los mecanismos utiliza para borrar de manera permanente del cluster los trabajos seleccionados. El usuario de la Web puede seleccionar varias filas de la tabla que muestra el historial de trabajos y eliminarlos. Lo que ocurre en el servidor es que se hace una separación de los trabajos por su estado y se elimina cada uno utilizando el método correspondiente de `Connections` al tiempo que son eliminados de la base de datos. La respuesta que se envía al navegador es la nueva tabla del historial sin los trabajos que han sido borrados.

- Descarga de archivos: la aplicación permite la descarga de los archivos de entrada o los resultados obtenidos, en caso de haber concluido la ejecución, de un trabajo seleccionado de la lista mostrada en el historial. Por decisión de diseño, sólo se pueden seleccionar trabajos de uno en uno para descargar sus archivos, de manera que las acciones `getInputs()` y `getResults()` inicien la descarga de los archivos. La arquitectura del sistema hace que la descarga de archivos sea algo más compleja que servir ficheros locales del servidor al navegador y viene acompañada de un potencial bloqueo, que como se explicó en el apartado 3.1.2, es conveniente evitar. La solución para este problema es la utilización de un sistema de actores que se explica con detalle más adelante.

- Compartición de trabajos: la Web incorpora la funcionalidad de compartir trabajos entre usuarios de la aplicación, de manera que el beneficiario de la compartición pueda tener acceso a los ficheros de entrada y salida de esos trabajos de los que no es propietario. Un usuario puede seleccionar de su historial uno o varios trabajos e introducir la dirección o direcciones de correo que identifiquen a los usuarios con los que quiere compartir ese trabajo. A partir de entonces se invoca la acción `shareJob()` que se encarga de separar los trabajos en función de si ya han sido compartidos antes o no y ejecutar, en función de ello, las acciones pertinentes. Si un trabajo ya está compartido sólo requiere añadir las instancias de tipo `Borrowed` a los usuarios con los que se va a compartir. En caso de no estar compartido es necesario descargar una copia de los archivos desde el cluster para ser guardados en el servidor, almacenando la ruta en la información del trabajo, como atributo de la entidad `InfoJob`. Cuando un trabajo está compartido se mostrará de

otro color en el historial y podrá ser compartido de nuevo así como seleccionar de una lista de los usuarios que tengan acceso con cuáles se quiere cesar la compartición mediante la acción `unshareJob()`.

- Creación del historial de trabajos del usuario: el historial de trabajos constituye la parte central de la zona privada ya que desde ella se tiene acceso a la mayor parte de funcionalidades que ofrece la aplicación. La manera en que la aplicación está desarrollada hace que todos los datos sobre los trabajos de un usuario estén en cada uno de los servidores a los que se accede. Para poder centralizar toda esa información es necesario hacer conexiones al servidor y hacer múltiples consultas: una por servidor al que tenga acceso y aplicación disponible en el host. Además, en la capa `Connections` se hace una diferenciación entre los trabajos ya terminados para los que se utiliza el método `Connections/ClusterConnection/getHistory()`, y los demás, para los que se usa `Connections/ClusterConnection/getRunningJobs()`. Todo esto es gestionado por la acción `consultarHistorial()` que recibe como parámetro un tiempo en milisegundos que indica si debe refrescar la tabla con información sobre el cluster o mostrar los datos contenidos en la base. Para evitar hacer todas estas conexiones que son necesarias para generar el historial mostrado al usuario, se guardan instancias de la lista de trabajos que lo componen y se van haciendo las modificaciones pertinentes, como borrar trabajos, actualizar su estado, etc. Sin embargo se proporciona un mecanismo para refrescar el historial con información del servidor cada media hora o forzando su actualización en caso de que haya habido cambios que por algún motivo no hayan sido registrados.

- Reordenación del historial: el historial se presenta en forma de tabla con un orden por defecto que muestra los trabajos ordenados por estado y fecha pero se da la posibilidad al usuario de que los ordene ascendentemente por otros campos; nombre, aplicación y PID en el servidor. Para hacerlo, la accion `reOrderTable()` es la encargada de guardar en la sesión el identificador que define el orden que el usuario ha seleccionado. El resto del trabajo se hace en la propia plantilla del historial que utiliza ese identificador para mostrar en el orden indicado los trabajos en la tabla.

- Consulta del estado de un trabajo: los trabajos que no están terminados pueden cambiar de estado durante la conexión del usuario. Cuando esto sucede, se necesita una acción que compruebe cuál es el estado actual del trabajo y lo modifique en la base de datos. De esta tarea se encarga `consultarTrabajo()` que comprueba la información de un trabajo a partir de consultas al servidor mediante conexiones con el paquete `Connections` y hace las modificaciones pertinentes en la base.

Esta acción forma parte del mecanismo de comunicación con el cluster mediante mensajes TCP que será desarrollado más adelante.

- `Admin.java`: para tener una gestión de la Web disponible para los administradores de la misma, se necesita un controlador que contenga las acciones relativas a su organización.

  - Añadir, eliminar o modificar usuarios de la Web: el administrador puede eliminar usuarios de la Web o modificar algunas de sus propiedades: principalmente, añadir nuevos accesos a servidores, introduciendo la identidad del usuario dentro del mismo.
  - Añadir, eliminar o modificar los servidores disponibles: los servidores disponibles en la aplicación pueden sufrir cambios a lo largo del tiempo por lo que se hace necesaria la posibilidad de modificar sus propiedades, sobretodo en lo que refiere a sus aplicaciones.

- Autenticadores: una de las herramientas que proporciona *Play!* es la composición de acciones. Se trata de la habilidad de hacer una cadena de múltiples acciones cada una de las cuales hace algo con la petición antes de pasarla a la siguente acción en la cadena. Esto resulta de gran utilidad a la hora de añadir autenticadores a nuestras acciones o controladores, que hagan una serie de comprobaciones antes de ejecutar la acción invocada con el objetivo de introducir restricciones de seguridad en el sistema. *Play!* viene con un autenticador incorporado, `Security.Authenticator`, del que basta heredar y sobreescribir sus métodos para adaptarlos a las características del autenticador que se necesite. En el caso de esta aplicación, al tener dos tipos de restricciones de acceso, se necesitan dos autenticadores:

  - `Secured.java`: correspondiente al acceso a la zona privada, comprueba si en la sesión actual HTTP está guardado el nombre del usuario que hace la petición: si lo está permite el acceso, de lo contrario redirecciona a la página de inicio de la aplicación.
  - `AdminSecured.java`: para el acceso a la zona de administrador es necesario comprobar que el usuario, cuyo identificador está registrado en la sesión HTTP, sea administrador mediante el atributo `isAdmin` de la entidad `User`. En caso negativo se redirecciona a la zona privada, en caso de que el usuario esté registrado en la sesion o a la página de inicio si ni siquiera ha iniciado sesión.

  Para utilizar estos autenticadores basta con anotar la acción que se desee restringir con `@Security.Authenticated(Secured.class)` en caso de usar el primero de los autenticadores y con `@Security.Authenticated(AdminSecured.class)` para el segundo.

> Cuando todas las acciones de un controlador requieren el uso de autenticadores, basta anotar la clase para que el autenticador afecte a todas las acciones contenidas. La división por ámbitos que se ha hecho de las acciones permite anotar con `Secured.class` el controlador `Jobs.java` y con `AdminSecured.class` el controlador de la parte administrativa, `Admin.java`.

Además de las clases de Java que pertenecen al paquete `controllers`, hay algunos ficheros adicionales que han de ser explicados en esta sección por el papel que juegan como soporte de algunas funcionalidades que no pueden ser satisfechas mediante el código ya descrito.
El fichero `app/Global.java` es una de estas clases, que implementa la interfaz del API de *Play!* `GlobalSettings`. En su método `onStart()` se puede incluir una parte del código que se ejecutará cada vez que se arranque la aplicación Web. Esto puede ser de gran utilidad en la fase de desarrollo para utilizar un conjunto de datos introducidos directamente en la base de datos usando YAML. Se utiliza la librería proporcionada por *Play!*, `play.libs.yalm`, para cargar datos almacenados en un fichero con la extensión `.yml` guardado en el directorio `/conf`. `Yaml.load(ïnitial-data.yml)̈`
Además de este uso en la fase de desarrollo, se ha utilizado para lanzar el servidor TCP necesario en la aplicación para recibir los mensajes del cluster al que se envía el trabajo. A diferencia de la aplicación original, en este caso no hay un servidor por cada usuario, sino que se establece una única conexión que permanece abierta desde el arranque y que recibe todos los mensajes del cluster que discrimina a partir del identificador único de trabajo que los mensajes deben incluir. Para abrir el servidor TCP se crea una instancia de la clase `Connections/TCPServer.java` y se utilizan dos de sus métodos, `discoverPort()` para encontrar un puerto libre en el que lanzar el servidor, y `getTcps()` para obtener un servidor TCP que poder arrancar. Metiendo todo este código en el método `onStart()` de esta clase `Global.java` se consigue arrancar el servidor una sola vez y poder usarlo con todos los usuarios que se conecten a la aplicación.
Las características y requisitos de la aplicación diseñada hacen que sea necesaria su comunicación con elementos externos al núcleo de la Web alojado en el servidor, cada una de las cuales necesita su propio mecanismo detallado a continuación.
La comunicación con el cluster es la más evidente ya que es la base de la aplicación pero tiene dos partes diferenciadas: la conexión abierta mediante `Connections/ClusterConnection.java` y el paso de mensajes desde el servidor a través del servidor TCP que acaba de ser explicado.
La comunicación con el paquete `Connections` puede hacerse de dos maneras: o bien incluyendo `.jar` en la declaración de librerías de la aplicación en el directorio `/lib` o incluyendo el código java en el paquete de controladores de manera que es compilado con el resto de ficheros. Durante el proceso de desarrollo se utilizó esta opción para solventar de manera más sencilla posible errores que forzasen la

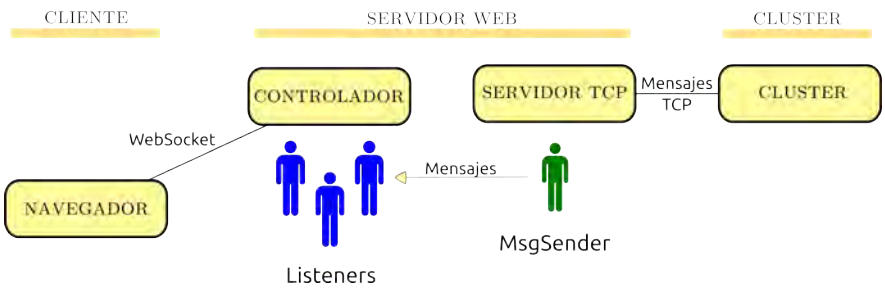


Figura 3.7: Sistema de actores para la comunicación TCP.

modificación del paquete original para adaptarlo a su uso en la Web. Sin embargo, para la fase de despliegue es conveniente añadir las librerías y hacer uso de ellas importándolas en las clases necesarias.
Por otro lado, en lo referente a la comunicación TCP es importante cubrir todos sus aspectos. La idea de comunicarse con el cluster mediante mensajes a través de un servidor TCP surge de la necesidad de recibir información de manera asíncrona del mismo. Cuando se lanza un trabajo, no se puede bloquear el sistema hasta que termine porque no sería eficiente. Por ello, se espera recibir un mensaje TCP que sirve como desencadenante de alguna acción en el servidor de la página Web. Cada uno de los mensajes recibidos sigue la estructura `<Instruccion>: <Identificador_trabajo><Usuario_del_host>` y los posibles valores de `<Instruccion>` son:

- *Submitted*: cuando el trabajo ha sido entregado al cluster.
- *Ended*: cuando el trabajo ha terminado su ejecución.
- *Aborted*: si ha habido algún problema y el trabajo cesa su ejecución antes de terminar.
- *Deleted*: cuando el trabajo ha sido eliminado del cluster.

El comportamiento asíncrono de este mecanismo requiere el uso de dos herramientas explicadas en la descripción de `Play!`: akka y *WebSockets*, el primero para que la llegada de mensajes al servidor TCP pueda desencadenar otras tareas en el servidor y el segundo para enviar el resultado de esas tareas al cliente. El sistema de actores diseñado tiene dos actores como se ve en la Figura 3.7: `MsgSender` y `Listener` que se comunican mediante mensajes para conseguir reaccionar a la recepción de los mensajes. El `MsgSender` es único para la aplicación y va ligado al propio servidor TCP: por eso se crea en el mismo momento en que se lanza éste, dentro del método `onStart()` en la clase `app/Global.java`. Además se guarda una referencia a ese actor de tipo `ActorRef` en el controlador `Application` para que esté accesible para la creación de los actores de tipo `Listener`. Su función

principal es la de recoger mensajes de tipo `Listen` que los actores `Listener` mandan tras su creación para dar a conocer que están activos y su referencia. Cada vez que el `MsgSender` recibe uno de estos avisos, guarda su referencia y el identificador del usuario al que pertenece; de esta forma, cuando se recibe un mensaje en el servidor TCP se puede mandar en *broadcast* a todos los actores `Listener` que hay registrados. El actor que se corresponda con el usuario al que va dirigido el mensaje es el que debe llevar a cabo la tarea correspondiente a la instrucción recibida. Las tareas posibles en función del contenido del mensaje son:

- Actualizar el estado y crear un mensaje de texto para mostrar una notificación en la Web para los casos de `Submitted`, `Ended` y `Aborted`.
- Eliminar el trabajo de la base de datos en caso de que aún estuviera en ella en el caso de `Deleted`. Este mensaje no se notifica al usuario y sólo se mantiene en caso de que, en el tiempo de conexión de un usuario a la Web, se elimine del sistema algún trabajo usando la aplicación de escritorio.

La comunicación con el cliente es esencial en este punto ya que el resultado generado en las tareas debe ser entregado. Para poder enviar el mensaje y otros identificadores, se convierte todo en un objeto de tipo JSON y se envía a traves de un *WebSocket*. Cada vez que se entra en la zona privada de la Web, desde el código del cliente se abre un *WebSocket* haciendo una llamada a la acción `msgTCPStream()` ya explicada, con el código que aparece el en Fichero 3.12. En ella se crea una instancia de `Listener` a la que se pasa el socket creado como atributo para que pueda pasar los resultados al cliente, donde se muestra la notificación correspondiente.

Fichero 3.12: Apertura del WebSocket en JS

```
open: =>
    @routeToStream = jsRoutes.controllers.Application.msgTCPStream(@now())
    @socket = new WebSocket(@routeToStream.webSocketURL())
    @socket.onmessage = (msg) =>
        data = $.parseJSON(msg.data)
        if data.Usuario is "true"
            if data.Comando is "Submitted" or data.Comando is "Ended"
                notif = new Notificacion
                    el: $("#over")
                notif.show(data.Mensaje)
```

Por último, como mejora del sistema se ha tenido en cuenta el modelo de actores como solución al bloqueo potencial que suponía la descarga de ficheros desde el cluster cuando estos son pesados, lo que es habitual en este tipo de trabajos de alta carga computacional. En la primera versión de la acción `getResults()` el proceso seguido era el siguiente:

1. Establecer una conexión al servidor al que pertenece el trabajo y configurarla con la aplicación correspondiente mediante los métodos `connect()` y `setAppName()` de la clase `Connections/ClusterConnection.java`.

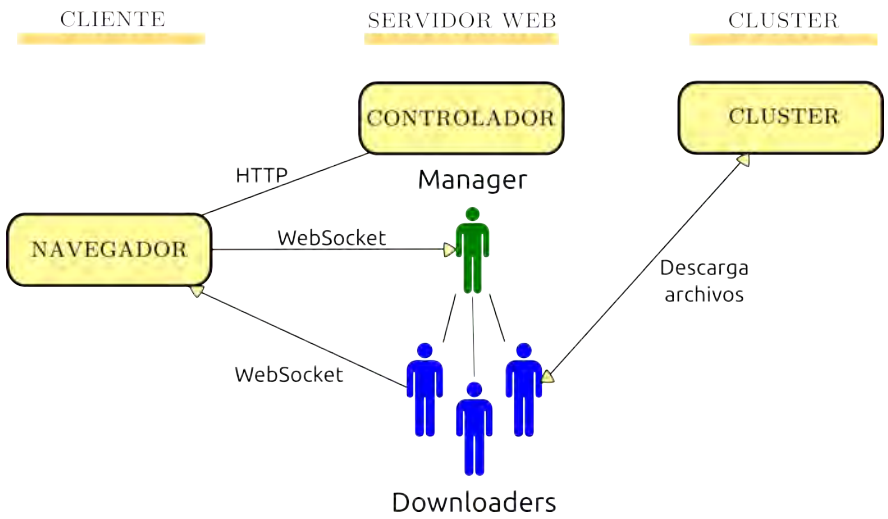


Figura 3.8: Sistema de actores para las descargas en segundo plano.

2. Solicitar la descarga de los archivos con el método `getResults()` determinando la ruta del directorio local en el que han de ser guardados. Esta parte es la que puede bloquear la aplicación en función del tamaño de los resultados a descargar.

3. Comprimir los resultados en un archivo `.zip`

4. Devolver los archivos como resultado de la acción como ya se detalló en la descripción de `Play!`.

Para poder eliminar el bloqueo en el servidor y de cara al cliente, quien debería poder seguir usando la Web mientras se descargan los ficheros, se parte la tarea y se introduce un sistema de actores para ejecutar la tarea de descargar archivos del cluster en un plano secundario. De nuevo se diseña un sistema de actores, representado en la Figura 3.8, con dos tipos de actores: `Manager`, del que sólo habrá una instancia creada en la clase `app/Global.java` y `Downloader`, con múltiples instancias que se crean cada vez que se abre el *WebSocket*. En este caso se utiliza la jerarquía de estos sistemas para resolver el problema. La comunicación entre actores de un sistema akka, se hace utilizando el paso de mensajes, para lo que se requiere una referencia de tipo `ActorRef` del actor al que se envía el mensaje. Para no tener que almacenar las referencias a todos los actores de tipo `Downloader` se aprovecha que cada actor tiene acceso a tres referencias en todo momento, la suya, la de su supervisor y la de sus hijos, como se muestra en la Figura 3.9. La utilización de un *WebSocket* vuelve a ser necesaria para comunicarse de manera asíncrona con el cliente pero es completamente análoga al caso del servidor TCP. En este caso, en lugar de crear una instancia en el propio código, se envía un mensaje al `Manager` para que cree un actor de tipo

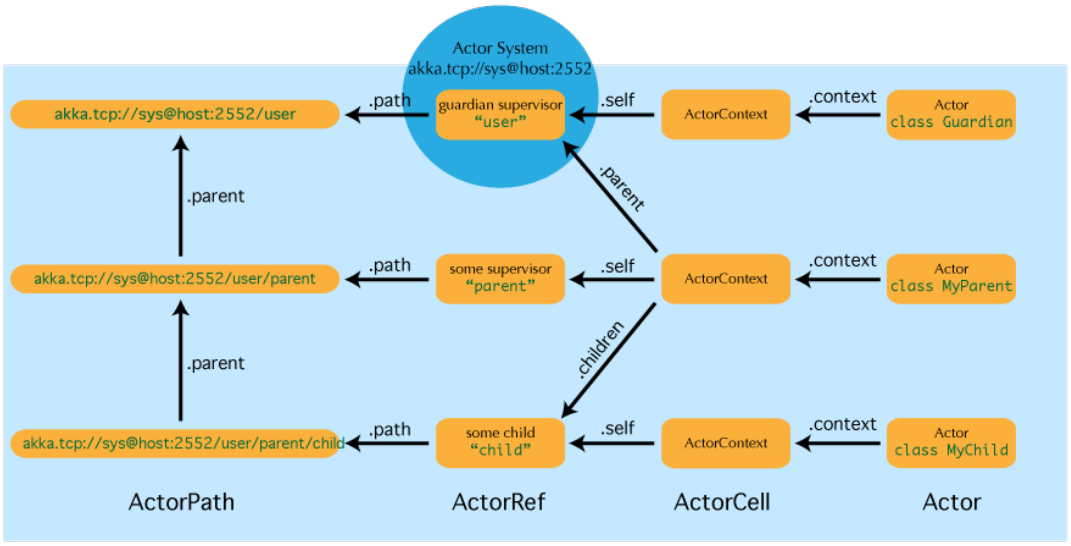


Figura 3.9: Relaciones entre actores en akka. Fuente: [4].

`Downloader` cuando se abre el socket y otro para que lo destruya cuando el socket se cierra.
El proceso para obtener los resultados de un trabajo pasa a ser el que sigue:

1. La acción `Jobs/getResults()` se encarga de enviar un mensaje al actor principal, `Manager`, con los datos de la descarga e inmediatamente devuelve su objeto `Result` evitando cualquier bloqueo. Cuando el `Manager` recibe un mensaje de este tipo, lo reenvía a todos sus hijos para que el que se corresponda con el usuario del que procede la petición comience la descarga de archivos desde el cluster de forma que se encarga de los pasos 1 y 2 del anterior proceso.

2. Cuando finaliza la descarga en el actor `Downloader`, este envía un mensaje a través del socket para notificar el fin de la descarga.

3. Esto desencadena en el código del cliente una llamada a la acción, `downFiles()`, que llevará a cabo los pasos 3 y 4 del proceso antiguo, acceder al fichero local en el que se han guardado los archivos, comprimirlos y enviarlos al cliente.

Hay tres funcionalidades de la aplicación que se benefician de este sistema para evitar el bloqueo: descarga de archivos de entrada, descarga de resultados y compartición de trabajos. Las tres comparten las acciones mencionadas para hacerlo, ya que el código discrimina los trabajos según la tarea que se aplica sobre ellos; sin embargo, la compartición de trabajos merece más consideraciones. Cuando un trabajo es compartido por primera vez es necesario descargar del cluster todos los archivos tanto de entrada como de salida y guardarlos de manera local

en el servidor de la aplicación Web para que estén disponibles para los demás usuarios que tienen acceso. Esto es debido a que de cara al cluster, se necesita ser propietario del trabajo para acceder a sus archivos y trasladándolos al servidor se solventa esa problemática. Basta actualizar la información del trabajo en la entidad `InfoJob` con las rutas del servidor en las que se encuentran descargados los archivos. Desde el punto de vista del usuario, el trabajo descargará archivos de la misma manera que si fuera propio, aunque un usuario no puede compartir ningún trabajo del que no sea propietario.

3.3.3. Vista

Este apartado incluye un recorrido por el físico de la aplicación: tanto la estructura de su contenido que corre a cargo del motor de plantillas, como su apariencia definida a partir de hojas de estilo tradicionales, CSS. El motor de plantillas, que ya fue explicado en el apartado 3.1.3, contiene los elementos que pueden verse a continuación en el árbol del directorio:

```
/views
├── admin
│   ├── addhost.scala.html
│   ├── item.scala.html
│   ├── showuser.scala.html
│   └── update.scala.html
├── adminboard.scala.html
├── dashboard.scala.html
├── indexlog.scala.html
├── indexreg.scala.html
├── index.scala.html
├── jobs
│   ├── addjob.scala.html
│   ├── historial.scala.html
│   ├── item.scala.html
│   └── myaccount.scala.html
├── login.scala.html
├── main.scala.html
└── register.scala.html
```

Están organizados desde lo más general a lo más específico ya que en el directorio raíz, `views`, se encuentran las plantillas correspondientes a las páginas principales:

- `index`: página de inicio de la aplicación que cuenta con dos formularios *pop up*, `login` y `register`, para el inicio de sesión y el registro de usuarios y con una descripción de las características de la aplicación.

- `indexlog` e `indexreg`: se trata de dos páginas de respaldo en caso de que el navegador no admitiera los formularios dinámicos o en caso de redirección

desde una acción, como ocurre cuando hay un fallo en el inicio de sesión.

- `main`: es el marco de la aplicación, que contiene todas las cabeceras necesarias para la zona privada y la zona del administrador.

- `adminboard` y `dashboard`: añaden a `main` el contenido para la zona del administrador y la zona privada respectivamente.

En los dos directorios restantes `jobs` y `admin` se almacenan las plantillas específicas utilizadas en `dashboard` y `adminboard`.
En cuanto a las hojas de estilo, se ha prescindido del uso de LESS puesto que el diseño es heredado parcialmente de la versión implementada usando JSP y Java Servlets por lo que se reutilizaron los ficheros añadiendo las modificaciones necesarias para adaptarse a la nueva apariencia.
La parte destacable de las hojas de estilo en este proyecto es que se han implementado dos diseños para poder adaptarse a los distintos dispositivos desde los que puede accederse a la página Web. Ambos diseños se muestran en imágenes en el capítulo 4. El principal de los diseños define la apariencia de la plataforma cuando se accede a ella a través de un ordenador, ya que el tamaño de la pantalla permite mostrar prácticamente toda la información de los trabajos de una forma cómoda. Sin embargo, para facilitar la navegación de la Web desde dispositivos móviles o tablets con una pantalla de menor tamaño se ha creado una versión simplificada que deja ver muchos menos elementos dando un aspecto más limpio a la zona privada. Las funcionalidades se mantienen idénticas en ambos casos pero la forma de mostrarlas varía con el fin de aportar una mejora frente a la aplicación para Android que tiene `Posidonia`, permitiendo a usuarios con otro sistema operativo una experiencia sencilla en su terminal.

3.3.4. Programación del lado del cliente: *CoffeeScript*

Como ya se ha explicado con anterioridad, la programación en el lado del cliente es esencial para dotar de dinamismo la plataforma y tener una comunicación completa con el usuario proporcionándole una navegación agradable por el sitio Web. Para la aplicación desarrollada en este proyecto se ha elegido hacer uso de la herramienta *CoffeeScript* cuyo funcionamiento está integrado en la infraestructura de *Play!*.
Como la cantidad de código que se requiere en el lado del cliente es considerable, conviene utilizar alguna herramienta más que simplifique la estructura del código.

- jQuery: ya se definió en el capítulo 2 y constituye una parte fundamental del código desarrollado ya que permite una manipulación completa y fácil del DOM. También se beneficia de su implementación de la técnica AJAX que permite hacer peticiones HTTP sin tener que recargar toda la página.

- Backbone.js, véase [58]: es una herramienta de desarrollo que surge como soporte para desarrollo de aplicaciones en el lenguaje JavaScript basada, de nuevo, en el paradigma MVC. Está pensada para aplicaciones de una única página, por lo que se ajusta al modelo seguido en la zona privada de *Posidonia*. Consiste en hacer una división de los elementos del DOM por clases que, en función de la parte del paradigma a la que pertenezcan, heredan de `Backbone.Model`, `Backbone.Collection`, o `Backbone.View`. De esta manera queda una estructura definida en la que podemos añadir un comportamiento a los elementos por secciones.

Aunque hay más ficheros de programación en el lado del cliente, el que merece una explicación detallada por ser el que contiene el grueso del código del lado del cliente es el fichero incluido en el documento HTML principal de la aplicación, `main.html`. Las clases más relevantes que contiene son:

- `Ventana`: gestiona la parte que muestra el contenido de la zona privada, es decir, el menú lateral y la ventana donde se muestra el contenido de cada opción del menú. Se encarga de inicializar todas las clases que gestionan el contenido de las distintas opciones del menú y de controlar los cambios de una opción a otra que son de dos tipos: desencadenados por un evento `click` del cliente en el menú o automáticos después de alguna acción como pasar del formulario de envío al historial cuando un trabajo es entregado al cluster.

- `Historial`: se corresponde con la parte central de la aplicación, el historial de trabajos del usuario. Se encarga de inicializar cada uno de los elementos de la tabla, de tipo `Job`. Tiene aproximadamente un método por cada una de las funcionalidades que ofrece la aplicación en relación a los trabajos, que se encargan por lo general de hacer llamadas AJAX a las acciones pertinentes y refrescar, en la mayor parte de los casos, la tabla después de su ejecución.

- `Job`: hace referencia a cada fila dentro del historial y se encarga de dotar de dinamismo la apariencia de la tabla. La selección de los trabajos y su sombreado al ser seleccionados son gestionados en esta clase.

3.4. Metodología de trabajo

A la hora de afrontar un proyecto amplio conviene establecer una estrategia de trabajo que garantice un orden para conseguir que resulte sencillo completar hitos más pequeños. En el caso del desarrollo de esta plataforma Web, se ha optado por una metodología que consiste en ir de casos simplificados de las funcionalidades que se pretendían alcanzar e ir escalando hacia la versión completa de los mismos. De esta manera se consigue tener siempre una versión funcional aunque no sea

óptima en caso de no conseguir completar alguno de los hitos por su elevada complejidad o porque surjan problemas en su desarrollo.
Las fases de este proyecto quedan detalladas a continuación:

- Esquema de la base de datos e implementación de las clases pertenecientes al paquete `app/models`.
- Programación del controlador `Application.java` con la funcionalidad básica de cualquier página Web: inicio y cierre de sesión y registro de nuevos usuarios.
- Incorporación del diseño básico con hojas de estilo CSS, heredado de la aplicación desarrollada con la tecnología JSP y Servlets de Java. En este punto se incluye la adaptación de los documentos HTML también heredados de este diseño anterior, a los que se introduce el uso de Scala y se convierten en plantillas propias de la arquitectura de *Play!*.
- Acciones provisionales. Antes de incorporar el paquete `Connections` de *Posidonia* se hace una versión de todas las funcionalidades básicas sin tener conexión real a la aplicación. De esta forma se consigue tener un esquema detallado de lo que será la aplicación completa sin necesidad de afrontar los problemas o dificultades que puede plantear la comunicación con el cluster. Para ello se obtienen los datos necesarios, en un futuro a devolver por la aplicación, de la base de datos, donde se pueden almacenar conjuntos de datos ficticios para hacer tests en la Web. Las acciones desarrolladas en este punto son:

  1. Enviar trabajos: guarda los datos en la base y almacena los archivos en una carpeta del servidor local. De esta manera se prueba el acceso a la base de datos y la transferencia de archivos al servidor Web.
  2. Mostrar trabajos en la tabla.
  3. Eliminar trabajos de la tabla.
  4. Lanzar el servidor TCP.

- *CoffeeScript* y AJAX: se dota a la Web de comportamiento dinámico y se añaden llamadas a través de AJAX a cada una de las acciones ya desarrolladas.
- Incorporación del paquete `Connections`. Se añade la comunicación real con el cluster sustituyendo uno a uno los métodos ficticios por otros con datos reales, en este caso se aplica la metodología propuesta siguiendo estos pasos:

  1. Introducir el paquete `Connections` en `app/controllers`.
  2. Añadir las dependencias necesarias para ejecutar el código de *Posidonia* y `JSCH.jar`.

3. Creación de acciones que utilizan las llamadas a los métodos del paquete `Connections` metiendo los datos diréctamente desde el código.
4. Probar las llamadas AJAX desarrolladas para las acciones ficticias con estas nuevas acciones.
5. Introducir formularios para usar datos provistos por el usuario para las acciones de comunicación con el cluster.

- Gestión de mensajes del servidor TCP. En lo relativo a la comunicación con el cluster a través de mensajes TCP, también aplica ese modelo de trabajo siendo las opciones desarrolladas las siguientes:

  1. *Poller*: se trata de una solución en la que se establece una cadencia para hacer consultas a la base de datos para poder reaccionar de una manera más o menos rápida a cambios en esta. En el caso de los mensajes, al llegar serían almacenados en la base de datos y detectados con un cierto retardo tolerable por el *poller* desarrollado para mostrar la notificación pertinente.
  2. *Scheduler*: esta solución es algo más eficiente en el uso de recursos pero sigue siendo un caso de consultas periódicas a la base de datos en busca de cambios.
  3. Sistema de actores: es la más compleja de las soluciones pero también la más eficiente con diferencia ya que sólo se ejecuta código cuando los mensajes han llegado y es inmediato, evitando el retardo introducido por las opciones anteriores.

- Descarga de archivos. Se hace primero sin tener en cuenta el tiempo de descarga que bloquea temporalmente los recursos de la Web y posteriormente se actualiza a la versión definitiva que utiliza un sistema de actores akka que permite continuar con la ejecución y ocuparse de las descargas en un segundo plano sin bloquear los recursos.

- Administrador de la Web. Se programa todo el código que permite al administrador, gestionar los datos más relevantes de la página sin tener que acceder de manera directa a la base de datos del sistema.

- Ultimar el diseño. Se termina el diseño a través de hojas de estilo de la página Web completa y se incorpora el diseño alternativo para adaptarse al acceso desde dispositivos móviles.

Capítulo 4

Descripción del sitio Web

Una vez detallada la arquitectura de la aplicación, corresponde a este capítulo hacer una descripción acompañada de imágenes de la funcionalidad completa de la plataforma en sus dos versiones: para ordenador y para dispositivos móviles.
Para que la descripción sea más ordenada y permita mantener una visión general de la plataforma se va a utilizar un caso de uso común de la página donde se exploren todas las posibilidades que ésta ofrece. El ejemplo de uso va desde el registro de usuarios y la utilización de todas las funcionalidades de *Posidonia*, hasta un recorrido por la parte de la Web destinada a la gestión disponible únicamente para los administradores de la misma. Con el objetivo de ofrecer un recorrido detallado y puesto que ambas versiones ofrecen la misma funcionalidad pero con distinta apariencia, el ejemplo de uso se aplica desde la versión móvil y desde la versión de ordenador en paralelo, permitiendo apreciar la adaptación realizada de una versión a otra.

4.1. Registro de usuarios e inicio de sesión

La página de inicio de la Web hace una presentación de las principales características de la aplicación y permite tanto el inicio de sesión como el registro de nuevos usuarios. Las Figuras 4.1 y 4.2 muestran esta pantalla de inicio con los botones que despliegan un *pop up* con cada uno de los formularios en el caso de la versión para escritorio, y redireccionan a la página de registro o inicio de sesión en el caso de la versión móvil. Cabe destacar que la versión de escritorio también tiene acceso a estas dos páginas de inicio de sesión y registro, a las que se redirige en caso de haber algún error al rellenar los formularios.
En el caso de uso utilizado para esta descripción, un usuario de nombre Adrián, abriría el formulario de registro introduciendo su cuenta de correo, nombre y contraseñas, como se ve en las Figuras 4.3. Si ha introducido correctamente todos los datos, es decir, con el formato correspondiente y la cuenta de correo no está registrada en la Web, se le redirige directamente a la zona privada. En caso contrario, se muestra la página de registro con un mensaje que indica el error

Figura 4.1: Página de inicio de la versión para ordenador.

Figura 4.2: Página de inicio de la versión para móvil.

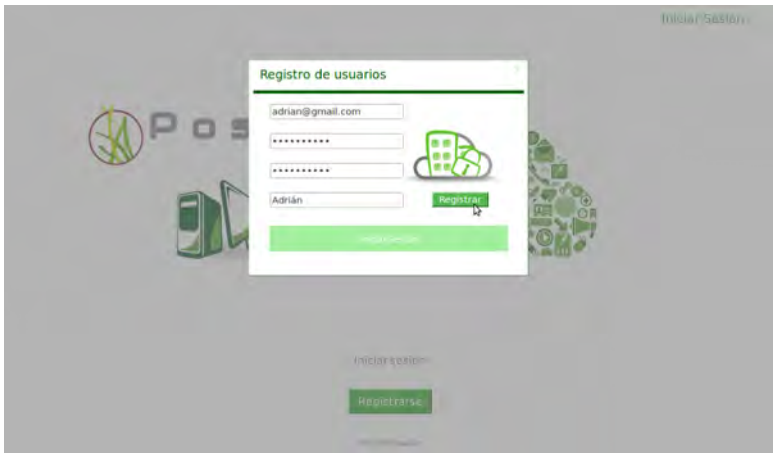


Figura 4.3: Registro de usuarios.

Figura 4.4: Página de error en el registro.

ocurrido y el formulario de nuevo, para volver a intentar el registro como aparece en la Figura 4.4. El inicio de sesión de un usuario ya registrado sigue el mismo procedimiento, teniendo que introducir únicamente el correo que le identifica en la Web y su contraseña y siendo el error posible que ese par usuario-contraseña no esté registrado.
A partir de ese momento, la sesión está asociada a ese usuario identificado por la dirección de correo que introdujo hasta que se cierre el navegador o la sesión de manera voluntaria.

Figura 4.5: Zona privada para un usuario recién registrado.

4.2. Gestión de la cuenta de usuario en la Web

Una vez en la zona privada de la aplicación, el usuario tiene una serie de posibilidades para gestionar su cuenta de usuario. En la Figura 4.5 se ve el aspecto inicial de la zona privada de un usuario recién registrado, que aún no tiene acceso a ningún servidor por lo que su historial se muestra vacío. Esta página contiene un menú vertical a la izquierda para la versión de escritorio y un menú horizontal en la parte superior para los dispositivos móviles buscando ahorrar espacio teniendo en cuenta los dispositivos con pantallas estrechas. Además tiene una ventana donde se muestra el contenido de cada una de las opciones del menú. En la versión de escritorio aparece el nombre y dirección de correo del usuario en la esquina superior derecha, junto con un botón de opciones que al ser pulsado despliega dos botones:

- *Mi cuenta*: selecciona la opción con el mismo nombre del menú y muestra su contenido en la ventana, desde donde el usuario puede llevar a cabo gestiones sobre su cuenta.

- *Cerrar sesión*: permite al usuario cerrar la sesión y le devuelve a la página de inicio de la Web.

En el caso de los dispositivos móviles, por cuestiones de espacio y por la disposición horizontal del menú se obvia el botón de *Mi cuenta* así como la dirección de correo del usuario y se cambia el botón de opciones por un botón para cerrar la sesión que tiene el mismo fucionamiento que *Cerrar sesión* como se aprecia en la Figura 4.6.
Una vez en la opción *Mi cuenta* del menú, el usuario tiene dos opciones mostradas en la Figura 4.7:

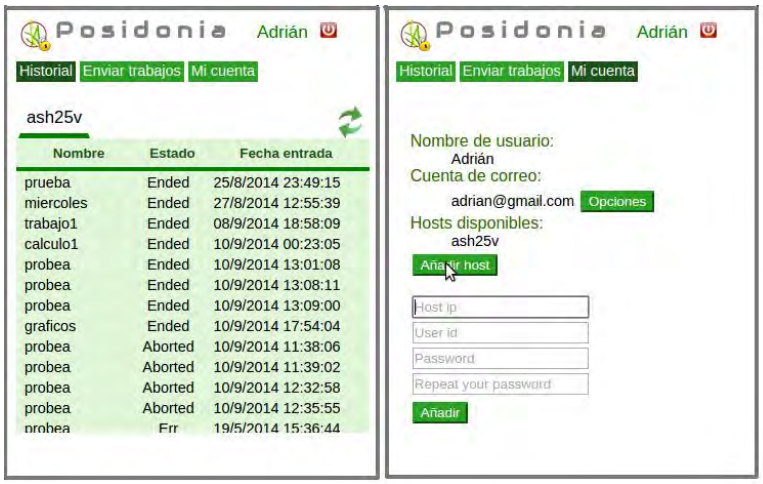


Figura 4.6: Zona privada para la versión móvil.

- Cambiar su contraseña: se le permite cambiar de contraseña introduciendo la contraseña antigua y la nueva dos veces por seguridad.
- Añadir acceso a servidores: como ya se explicó en el capítulo 3, el acceso a cada uno de los servidores es independiente y cuenta con un par usuario-contraseña. La plataforma Web facilita al usuario que tenga acceso a varios servidores, gracias a la concentración de todos en una misma cuenta de usuario Web. Para ello el usuario debe introducir la dirección IP del servidor y su usuario y contraseña de acceso al mismo. Este proceso se hace una única vez: a partir de entonces, la aplicación recogerá de forma simultánea la información de todos los servidores a los que un usuario tiene acceso presentando toda la información, separada en pestañas por cada uno de los servidores. Cuando se introduce un par usuario-contraseña para un servidor determinado se hace una comprobación, intentando la conexión con el mismo. Si la conexión es satisfactoria se añaden los datos a la cuenta del usuario, de lo contrario no se modifica.

En el ejemplo de este capítulo, el usuario introduce sus credenciales para el servidor con la dirección `192.168.151.203` (alias: `ash25v`).

4.3. Envío de un trabajo

Para ilustrar el envío de trabajos al cluster se utiliza una tarea simple en Matlab que resulta suficiente para ver el su comportamiento genérico. El código que se quiere enviar al cluster cuenta con las siguientes entradas:

- Un *script*, `prueba.m`, mostrado en el Fichero 4.1, que contiene una serie de comandos que incluyen llamadas a otras funciones `.m` con parámetros

Figura 4.7: Opciones de *Mi cuenta* para la versión de ordenador.

de entrada y salida, y un archivo `.mat` donde se guardan algunas variables junto con sus valores.

- Funciones de Matlab llamadas por el *script*, `cuadratica.m` y `lineal.m`.
- Un contenedor de variables llamado `plots.mat`.

Las salidas que este código genera al ser ejecutado son:

- Imágenes con extensión `.jpg` correspondiente a las representaciones llevadas a cabo en el script mediante el comando `plot`, guardadas en el Fichero 4.1 como: `plot1.jpg`, `plot2.jpg`, `polar.jpg`, `plot3d.jpg`, `cont.jpg` y `plot3d2.jpg`.
- Un contenedor de variables, `script2.mat`, en el que se almacenan todas las variables y sus respectivos valores al acabar la ejecución. En el caso de la ejecución remota en un cluster es clave este contenedor para no perder los resultados y que puedan ser devueltos junto con las imágenes generadas.

Los pasos que hay que seguir para enviar una tarea como la descrita son:

1. Seleccionar en el menú, vertical para el caso de la versión para ordenador y horizontal para la móvil, la pestaña *Enviar trabajo* que muestra una serie de campos a rellenar y seleccionar que determinarán las características del envío y son los que siguen y pueden verse en la Figura 4.8:
    - Nombre del trabajo: es el nombre que identificará al trabajo en la plataforma. Gracias a la utilización del identificador único heredado de la aplicación original, *Posidonia*, se permite tener múltiples trabajos con el mismo nombre.

Fichero 4.1: prueba.m

```
%Se usa el contenedor.
load plots.mat

x = lineal(1:10);
y = cuadratica(1:10);

%Algunas representaciones.
h = plot(x,y);

saveas(h, 'plot1.jpg');

h2 = plot(x,x);
saveas(h2, 'plot2.jpg');

t = 0:.01:2*pi;

h3 = polar(t,sin(2*t).*cos(2*t),'--r');
saveas(h3, 'polar.jpg');

th = (0:127)/128*2*pi;

x = cos(th);

y = sin(th);

f=abs(fft(ones(10,1),128));

h4 = stem3(x,y,f');
saveas (h4, 'plot3d.jpg')

[X,Y] = meshgrid(-2:.2:2,-2:.2:3);

Z = X.*exp(-X.^2-Y.^2);

[c, h5] = contour(X,Y,Z);
saveas(h5, 'cont.jpg')

[X,Y] = meshgrid(-3:.125:3);

Z = peaks(X,Y);

h6 = mesh(X,Y,Z);
saveas(h6, 'plot3d2.jpg')

save script2.mat
```

Figura 4.8: Formulario de envío de trabajos.

- Servidor: un desplegable que contiene todos los nombres de los *hosts* a los que el usuario tiene acceso para que seleccione a cuál envía la tarea.
- Nombre de la aplicación: es otro desplegable que permite elegir de entre todas las aplicaciones posibles aquellas que están disponibles en el servidor seleccionado. La disponibilidad de las opciones se modifica dinámicamente con la selección del servidor de manera que, aunque aparecen todas las aplicaciones, el desplegable sólo permite seleccionar aquellas disponibles, marcadas en un color más oscuro.
- Número de *slots*: se permite la elección del número de *slots* que pueden ser utilizados para la ejecución del trabajo enviado.
- Archivos de entrada: consiste en un elemento *file chooser* para la selección de los archivos de entrada del trabajo. En el caso del ejemplo se seleccionan los archivos `.m` y `.mat`.
- Nombre del *script*: de los archivos con extensión `.m` hay que especificar el nombre del *script* a ejecutar en el cluster, sin la extensión.

2. Los campos de este formulario son obligatorios, no se puede ejecutar el envío sin que todos estén completados. Tras rellenarlos basta con hacer *click* en el botón de *Enviar*.

3. Tras el envío se cambia de manera automática a la pestaña *Historial* a la espera de recibir las notificaciones pertinentes:

   - Notificación de entrega del trabajo al cluster: modifica el estado del trabajo a `R` (Running).

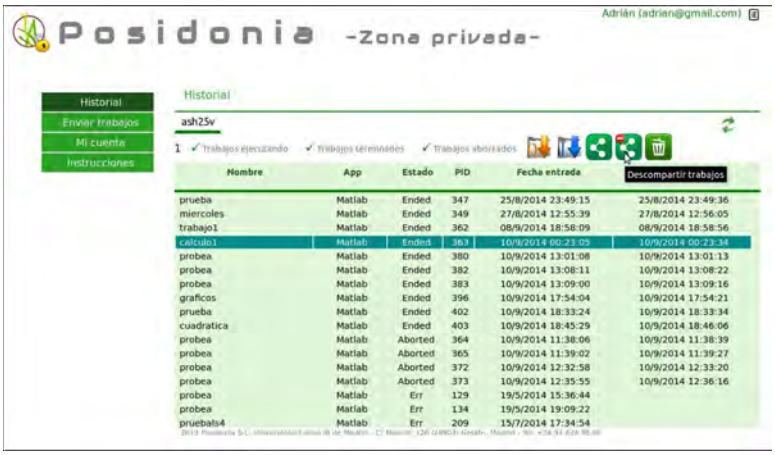


Figura 4.9: Opciones de *Historial* para la versión de ordenador.

- Nofificación de finalización del trabajo enviado: actualiza el estado del trabajo a `Ended` y añade la fecha de finalización.
- Notificación de trabajo abortado: si surge algún problema en la ejecución y es interrumpido se recibe esta notificación que actualiza el estado del trabajo a `Aborted`.

La recepción de estas notificaciones es completamente asíncrona ya que se trata de los mensajes recibidos por el servidor TCP, explicado en el Capítulo 3. De esta forma el usuario puede seguir utilizando la Web y cuando se recibe la notificación se actualizan los elementos pertinentes.

4.4. Historial de trabajos

La opción *Historial* permite el acceso al resto de funcionalidades importantes de la aplicación referentes a los trabajos enviados. En ella se presenta una tabla con los trabajos del usuario agrupados por distintas pestañas según el servidor al que pertenezcan. El comportamiento dentro de cada pestaña es el mismo para todas ellas e independiente uno de otros.
El modelo de tabla varía para cada una de las versiones, siendo más extensa en la de escritorio, mostrada en la Figura 4.9, porque se cuenta con un mayor espacio de partida y puede mostrar todos los parámetros asociados a cada uno de los trabajos: nombre, estado, aplicación a la que pertenece, PID, fecha de entrada y fecha de salida. Para la versión reducida se mantienen únicamente el nombre, el estado y la fecha de entrada que resultan suficientes para identificar los trabajos que se quiere seleccionar, veáse la Figura 4.10.

- Actualización desde el cluster. Con el fin de que la Web sea lo más autónoma posible, se reduce al máximo el número de consultas del historial de

Figura 4.10: Opciones de *Historial* para la versión móvil.

trabajos a los servidores y se actualizan los datos ya contenidos en la base de datos. Sin embargo, para garantizar la validez de los mismos se hace una actualización periódica automática desde el cluster y se incorpora un botón en la parte superior derecha de la ventana *Historial* para que el usuario pueda, si lo desea, forzar la actualización.

- Filtrado de trabajos según el estado. Dentro de la tabla para cada servidor hay tres botones que permiten ver u ocultar los trabajos con determinado estado (en ejecución, terminados y abortados), permitiendo hacer un filtrado customizado de los trabajos disponibles. Esta funcionalidad sólo está disponible en la versión de escritorio ya que consume bastante espacio para las pantallas de los dispositivos más pequeños.

- Ordenación de trabajos con distintos criterios. Las seis columnas de la versión ampliada de la tabla que se citaron antes, permiten ordenar la tabla de forma ascendente en base a ese parámetro. La ordenación por defecto es en base al estado del trabajo colocándolos por el siguiente orden: trabajos terminados, trabajos en ejecución y trabajos abortados.

- Descarga de archivos de entrada y salida. La tabla permite la selección de uno o más trabajos de la tabla dentro de la misma pestaña del servidor al que corresponden. Las opciones disponibles aparecen cuando se seleccionan los trabajos y varían según el número de trabajos y sus caracerísticas. Las tres posibilidades son:

    1. Selección de un único trabajo no compartido: muestra las opciones de descargar archivos de entrada y salida, compartir y borrar la tarea del cluster.

2. Selección de un único trabajo compartido: muestra todas las opciones posibles, descarga de archivos, compartición, descompartición y eliminación del trabajo.
3. Selección de múltiples trabajos: este caso sólo da la opción de compartir y de eliminar los trabajos seleccionados, las únicas acciones aplicables a más de un trabajo al mismo tiempo.

Si el usuario del ejemplo, Adrián, selecciona de la tabla el trabajo que envió, `Prueba`, y hace *click* en descargar los resultados, obtendrá un mensaje de notificación indicando que la descarga puede llevar unos minutos, en función de su tamaño. Después el navegador mostrará las opciones estándar para la gestión de los archivos descargados. El contenido completo de los resultados del trabajo, al igual que pasa con los archivos de entrada, se descarga comprimido en un fichero `.zip`.

- Compartición de trabajos. Cuando un usuario quiere compartir uno o varios de sus trabajos basta con seleccionar estos de la tabla y elegir la opción *compartir* que aparece en el menú de opciones. Si uno de los trabajos de la selección no pudiera ser compartido porque el usuario no es su propietario, se elimina automáticamente de la selección y se muestra un mensaje en la ventana emergente que aparece al hacer *click* en *compartir*, veáse la Figura 4.11. Es entonces cuando el usuario puede introducir el correo electrónico de los usuarios con los que quiere compartir todos los trabajos que seleccionó. Si quisiera discriminar esta compartición por trabajos debe seleccionarlos individualmente. Cuando se comparte un trabajo, como ya se explicó en el capítulo 3, se descargan del cluster los archivos de entrada y salida para ser almacenados en el servidor y que los usuarios tengan acceso a ellos sin necesidad de tener clave en el cluster al que pertenecen. Esto se hace en segundo plano, pero se notifica al usuario el comienzo y el fin de la descarga de los archivos indicando que todo ha funcionado correctamente.

- Descompartición de trabajos. Cuando se selecciona un trabajo que ha sido previamente compartido, se habilita la opción de deshacer esa compartición. El motivo por el que sólo se permite un trabajo cada vez es que se permite elegir los usuarios a los que se quiere eliminar el acceso al trabajo de todos los que lo tienen y cada trabajo puede estar compartido con un grupo distinto. Esta funcionalidad también abre una ventana emergente donde se listan todos los usuarios con acceso al trabajo. La lista permite seleccionar cada uno de los elementos, de forma que sólo se deshace la compartición para aquellos usuarios de la lista que estén seleccionados cuando se acepta la descompartición. En el caso de que todos los usuarios sean seleccionados para eliminar la compartición, los archivos descargados al servidor desde el cluster son eliminados.

Figura 4.11: Compartición de varios trabajos y mensaje de error.

- Eliminación de trabajos. Se permite al usuario la selección de múltiples trabajos para ser eliminados todos al mismo tiempo. Para evitar eliminar trabajos accidentalmente se incluye una ventana de confirmación antes de proceder a borrar los trabajos tanto de la base de datos como del cluster en el que fueron ejecutados. Es posible eliminar trabajos en ejecución, así como trabajos compartidos o que no pertenezcan al usuario, simultáneamente, ya que es la aplicación la que elige en función del tipo de trabajo el método de eliminación que utiliza. Cuando son eliminados, los trabajos desaparecen de la tabla.

4.5. Administrador

Cuando un usuario es administrador, en su zona privada aparece un enlace, *Administrador*, que le redirecciona a la zona de gestión de la página Web. La apariencia de esta parte, que se muesta en la Figura 4.12, sigue el diseño de la zona privada y sólo se encuentra disponible para la versión de escritorio. Con el objeto de evitar la manipulación directa de la base de datos, se han encapsulado en esta página una serie de funcionalidades básicas para gestionar la Web.

- Tabla de usuarios: permite eliminar usuarios (varios a la vez) de la Web con sólo seleccionarlos en la tabla y hacer *click* en *Eliminar*. Además permite hacer una seleción de un usuario particular cuyos datos relativos a la Web se precisen modificar, pasando de manera automática a la segunda de las opciones del menú.

- Actualización de usuarios: se debe seleccionar un usuario, bien desde la tabla como se ha explicado en el punto anterior, o bien introduciendo en

Figura 4.12: Zona de administración de la Web.

Figura 4.13: Gestión de servidores en la página Web.

esta pestaña la dirección de correo que lo identifica. En cualquiera de los casos se presenta la información del usuario de la cual se puede modificar el acceso a los servidores.

- Actualización de servidores: esta última funcionalidad, que se puede ver en la Figura 4.13, permite añadir los servidores a los que se puede tener acceso desde la plataforma para que sean registrados en la página Web y almacenados en la base de datos junto con las aplicaciones que tienen disponibles. Esto es importante ya que el formulario de envío de trabajos sólo permite la selección de las aplicaciones que estén definidas como disponibles en cada uno de los servidores.

Capítulo 5

Conclusiones y líneas futuras

En este proyecto se ha llevado a cabo la implementación de una plataforma Web para dar soporte a la aplicación *Posidonia* para el envío de tareas de diferentes aplicaciones a un cluster de computación científica facilitando su uso de cara al usuario.
Los objetivos que han sido cumplidos en dicha implementación son:

- Funcionalidad de la aplicación original: se ha conseguido introducir en la plataforma Web la funcionalidad completa de *Posidonia*, detallada en el capítulo 1.

- Ampliación de la funcionalidad: debido al carácter más dinámico de esta versión de la aplicación, se han introducido una serie de complementos como la compartición de trabajos con otros usuarios de la plataforma, o la posibilidad para el usuario de agrupar todos los accesos a servidores en una misma cuenta. De esta manera se consigue una experiencia mucho más sencilla para el usuario que puede gestionar todos sus trabajos en paralelo desde una misma sesión.

- Gestión de la página Web: se ha introducido una sección para que los administradores de la Web sean capaces de tener acceso a datos de gestión de la plataforma y su modificación de manera simple y evitando los accesos directos a la base de datos.

- Movilidad: dotar a la aplicación *Posidonia* de una movilidad casi total era la principal de las motivaciones ya que constituye una parte imprescindible en la actualidad si se quiere llegar a los usuarios. Establecer una plataforma que se adapta al dispositivo que se utilice para acceder a ella consigue evitar el desarrollo de distintas versiones específicas para terminales móviles garantizando no sólo la movilidad sino su carácter multiplataforma.

- Escalabilidad: la elección de *Play!* para el desarrollo de la plataforma permite que la escalabilidad de este proyecto sea sencillo debido a las propias

características de esta infraestructura que cuenta con una arquitectura muy modular e incorpora el uso de tecnologías que garantizan una aplicación simple del código, como Scala.

- Seguridad: se ofrece al usuario la transferencia segura de sus archivos aprovechando el protocolo de seguridad HTTPS que permite que tanto la navegación por la página Web como la comunicación de datos y archivos con la misma se realicen de manera segura. Además utiliza las librerías de *Posidonia*, lo que garantiza la seguridad entre el servidor Web y el *cluster* con los protocolos SSH y SCP, de forma que todos los elementos del escenario propuesto quedan cubiertos.

En definitiva, se han conseguido alcanzar los objetivos marcados para este proyecto dando lugar a una plataforma que hace posible expandir una aplicación ya existente, facilitando su acceso y actualización a partir de la centralización en un servidor Web.
Este es un proyecto que puede ser ampliado de varias maneras, por lo que se abren una serie de líneas futuras con el fin de mejorar sus características.

- Ampliación del formulario del envío de trabajos, mostrado en la Figura 4.8, para añadir opciones avanzadas de la ejecución de la tarea en el cluster.
- Compartición de trabajos vía correo electrónico para usuarios que no disponen de cuenta en la plataforma.
- Diseño e implementación de una pasarela que permita el envío de trabajos a otro tipo de clusters de alto rendimiento como el proporcionado por Amazon [59].

Capítulo 6

Presupuesto

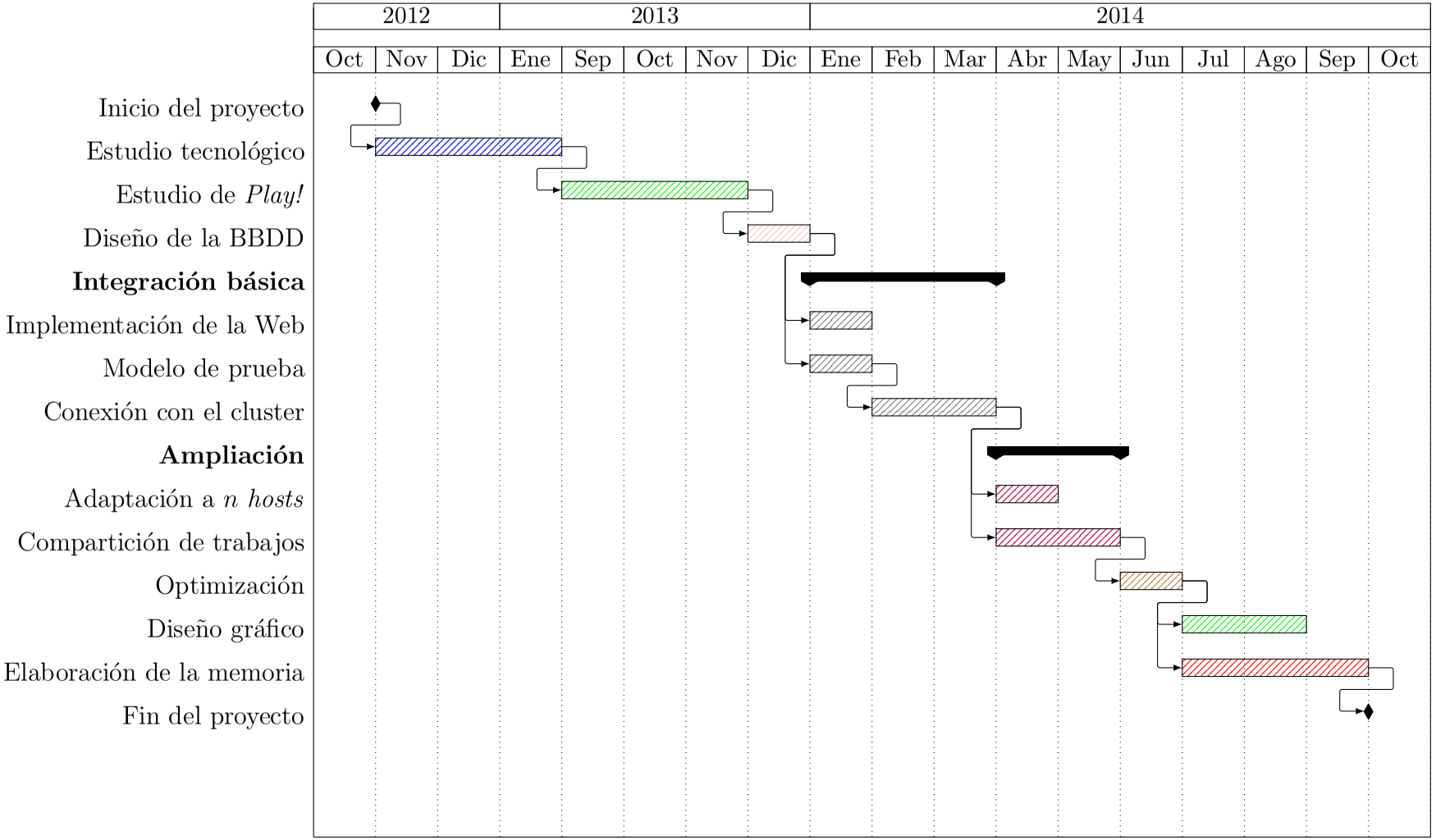
A lo largo de esta memoria se ha descrito todo el proceso y arquitectura de la plataforma Web desarrollada para satisfacer la necesidad de dotar de mayor movilidad a una aplicación ya existente, *Posidonia*, que permite facilitar a los usuarios el acceso a clusters de altas prestaciones.
El desarrollo completo de este proyecto se ha dividido en las siguientes fases:

- Estudio del estado del arte del diseño de plataformas Web y realización de prototipos de la misma en las tres alternativas más relevantes: PHP, JSP y *Play! framework*, expuesto en el Capítulo 2.
- Profundización en la herramienta elegida, *Play! framework*, llevando a cabo un aprendizaje de los nuevos lenguajes de programación necesarios para la realización del proyecto.
- Diseño de la base de datos e implementación de las clases contenidas en el paquete `models`.
- Integración en la plataforma Web de la funcionalidad básica de la aplicación original *Posidonia*. Esta fase puede dividirse, a su vez, en subfases:
  - Implementación de las características básicas de una Web con acceso restringido.
  - Primera implementación de las funcionalidades haciendo uso de un modelo de prueba, sin conexiones reales con el cluster.
  - Implementación definitiva de las funcionalidades de *Posidonia*, haciendo conexiones reales al cluster.
- Extensión de la funcionalidad de la plataforma, que tiene dos partes principales:
  - Adaptación de la Web a la conexión de varios servidores, *hosts*, por cada uno de los usuarios.

    - Compartición de trabajos.

- Optimización del uso de recursos del servidor por parte de la Web, en la que se incluyen los sistemas de actores akka para evitar los bloqueos producidos por la descarga de archivos desde el cluster.
- Diseño de la interfaz gráfica tanto para la versión móvil como para la versión para ordenador.
- Elaboración de la memoria.

Para detallar de manera cronológica el desarrollo de este proyecto se incluye un diagrama de Gantt donde aparecen todas estas fases.


2012
2013
2014
Oct
Nov
Dic
Ene
Sep
Oct
Nov
Dic
Ene
Feb
Mar
Abr
May
Jun
Jul
Ago
Sep
Oct
Inicio del proyecto
Estudio tecnológico
Estudio de *Play!*
Diseño de la BBDD
Integración básica
Implementación de la Web
Modelo de prueba
Conexión con el cluster
Ampliación
Adaptación a *n hosts*
Compartición de trabajos
Optimización
Diseño gráfico
Elaboración de la memoria
Fin del proyecto

Y finalmente, el presupuesto considerando todos los costes posibles (personas involucradas en el desarrollo, costes indirectos, equipos empleados, etc) es el que se muestra a continuación:

UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior

PRESUPUESTO DE PROYECTO

1.- Autor:

Cristina García Muñoz

2.- Departamento:

Teoría de la Señal y de las Comunicaciones

3.- Descripción del Proyecto:

- Titulo: **Plataforma Web de Simulación Remota en un Cluster de Computación Científica**
- Duración (meses): **17 meses**
- Tasa de costes Indirectos: **20%**

4.- Presupuesto total del Proyecto (valores en Euros):

39.841,00 Euros

5.- Desglose presupuestario (costes directos)

PERSONAL

Apellidos y nombre	N.I.F. (no rellenar, sólo a título informativo)	Categoría	Dedicación (hombres mes) [a]	Coste hombre mes	Coste (Euro)	Firma de conformidad
Cristina García Muñoz		Ingeniero Junior	10	2.450,00	24.500,00	
Adrián Amor Martín		Ingeniero Junior	2	2.450,00	4.900,00	
Ignacio Martínez Fernández		Administrador Senio	0,5	2.600,00	1.300,00	
		Hombres mes 12,5		**Total**	30.700,00	

[a] 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)
Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

EQUIPOS

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable [d]
Portátil	840,00	100	17	48	297,50
Cluster del Grupo de Radiofrecuencia	163.498,00	10	5	48	1.703,10
Servidor Dedicado	4.800,00	50	10	48	500,00
				Total	2.500,60

[d] Fórmula de cálculo de la Amortización:

A = nº de meses desde la fecha de facturación en que el equipo es utilizado
B = periodo de depreciación (60 meses)
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto (habitualmente 100%)

SUBCONTRATACIÓN DE TAREAS

Descripción	Empresa	Coste imputable
	Total	0,00

OTROS COSTES DIRECTOS DEL PROYECTO[e]

Descripción	Empresa	Costes imputable
	Total	0,00

[e] Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

6.- Resumen de costes

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	30.700
Amortización	2.501
Subcontratación de tareas	0
Costes de funcionamiento	0
Costes Indirectos	6.640
Total	39.841

Así, el presupuesto total de este proyecto asciende a la cantidad de 39.841 euros.
Leganés, a 3 de Octubre de 2014.
El ingeniero proyectista.

Glosario

A

AJAX *Asynchronous JavaScript And XML*, pág. 22.

API *Application Programming Interface*, pág. 4.

C

CPU *Central Processing Unit*, pág. 39.

CSS *Cascading Style Sheet*, pág. xiii.

CSS2 *Cascading Style Sheets v2*, pág. 20.

CSS3 *Cascading Style Sheets v3*, pág. 20.

D

DDL *Data Definition Language*, pág. 51.

DML *Data Manipulation Language*, pág. 52.

DOM *Document Object Model*, pág. 16.

DRM *Distributed Resource Management*, pág. 5.

DRMAA *Distributed Resource Management Application API*, pág. 4.

H

HPCC *High Performance Computing Cluster*, pág. 1.

HTML *HyperText Markup Language*, pág. xiii.

HTML5 *HyperText Markup Language v5*, pág. 16.

HTTP *HyperText Transfer Protocol*, pág. xiii.

HTTPS *HyperText Transfer Protocol Secure*, pág. 96.

I

IMAP *Internet Message Access Protocol*, pág. 24.

IP *Internet Protocol*, pág. 57.

J

JAR *Java ARchive*, pág. 42.

Java EE *Java Enterprise Edition*, pág. 33.

JDBC *Java DataBase Connectivity*, pág. 52.

JDK *Java Developement Kit*, pág. 29.

JPA *Java Persistence API*, pág. 52.

JPQL *Java Persistence Query Language*, pág. 53.

JRE *Java Runtime Enviroment*, pág. 29.

JS *JavaScript*, pág. 21.

JSch *Java Secure Channel*, pág. 7.

JSE *Java Standard Edition*, pág. 29.

JSON *JavaScript Object Notation*, pág. 46.

JSP *JavaServer Pages*, pág. XIII.

L

LDAP *Lightway Director Access Protocol*, pág. 24.

LESS *Extension of CSS*, pág. 41.

M

MVC *Model View Controller*, pág. XIII.

N

NNTP *Network News Transport Protocol*, pág. 24.

O

ORM *Object Relational Mapper*, pág. 54.

P

PDF *Portable Document Format*, pág. 24.

PHP *Hypertext Preprocessor*, pág. xiii.

PHP-GTK *PHP GIMP ToolKit*, pág. 24.

PID *Process IDentifier*, pág. 61.

POP3 *Post Office Protocol v3*, pág. 24.

R

RAM *Random-Access Memory*, pág. 61.

REST *Representational State Transfer*, pág. 36.

S

SBT *Scala Buidl Tool*, pág. 42.

SCP *Secure Copy*, pág. 3.

SGE *Sun Grid Engine*, pág. 4.

SNMP *Simple Network Management Protocol*, pág. 24.

SQL *Structured Query Language*, pág. xiii.

SSH *Secure SHell*, pág. 3.

T

TCP *Transmission Control Protocol*, pág. 7.

U

URI *Uniform Resource Identifier*, pág. 44.

URL *Uniform Resource Locator*, pág. 17.

X

XHTML *eXtensible HyperText Markup Language*, pág. 24.

XML *Extensible Markup Language*, pág. 24.

Y

YAML *YAML Aint́ Markup Language*, pág. 41.

Bibliografía

[1] A. Amor-Martín, I. Martínez-Fernández, D. García-Doñoro, and L. E. García-Castillo, "Herramienta de simulación remota en un cluster de computación científica," *URSI 2012*, Sep. 2012.

[2] "Arquitectura multinivel," http://www.infor.uva.es/~jvegas/cursos/buendia/pordocente/node21.html, Aug. 2014.

[3] "Documentación Play! framework en castellano," http://playdoces.appspot.com/documentation/1.2.3/main, Aug. 2014.

[4] "Documentación akka para Java," http://doc.akka.io/docs/akka/2.3.4/java.html?_ga=1.140996213.69048327.1400928693, Aug. 2014.

[5] A. Amor-Martín, "Herramienta de simulació remota en un cluster de computación científica." Master's thesis, Universidad Carlos III de Madrid, 2012.

[6] "Página oficial de Android," http://www.android.com/, Sep. 2014.

[7] "Manual de SCP," http://linux.die.net/man/1/scp, Sep. 2014.

[8] "RFC de SSH," http://www.ietf.org/rfc/rfc4251.txt, Sep. 2014.

[9] *Begginer's Guide to Sun Grid Engine 6.2*, 2008.

[10] "Condor Project," http://research.cs.wisc.edu/condor/, Sep. 2014.

[11] "Grupo de trabajo de DRMAA," http://www.drmaa.org/, Sep. 2014.

[12] "Página de la librería JSch," http://www.jcraft.com/jsch/, Sep. 2014.

[13] "Tutorial de JavaScript," http://www.w3schools.com/js/, Sep. 2014.

[14] "Sitio oficial de HTML," http://www.w3.org/html/, Aug. 2014.

[15] "Sitio oficial de CSS versión 1," http://www.w3.org/TR/CSS1/, Aug. 2014.

[16] "Especificación de HTML5," http://www.w3.org/TR/html5/, Aug. 2014.

[17] "Especificación de DOM," http://www.w3.org/DOM/, Aug. 2014.

[18] "Sitio oficial de ADOBE FLASH," http://www.adobe.com/es/products/flashplayer.html, Aug. 2014.

[19] "Sitio oficial de CSS versión 3," http://www.w3.org/Style/CSS/current-work, Aug. 2014.

[20] "Sitio oficial de CSS versión 2," http://www.w3.org/TR/CSS2/, Aug. 2014.

[21] "Sitio oficial de jQuery," http://jquery.com, Aug. 2014.

[22] "Introducción a AJAX," http://www.w3schools.com/ajax/ajax_intro.asp, Aug. 2014.

[23] "Sitio oficial de Java en castellano," http://www.java.com/es/, Jul. 2012.

[24] "Manual en castellano de PHP," http://php.net/manual/es/, Aug. 2014.

[25] "GTK+ project," http://www.gtk.org/, Aug. 2014.

[26] "Sitio oficial de MYSQL," http://www.mysql.com/, Aug. 2014.

[27] "PHP-Java Bridge," http://php-java-bridge.sourceforge.net/pjb/, Aug. 2014.

[28] "Sitio oficial de Tomcat," http://tomcat.apache.org/, Aug. 2014.

[29] "Uso de lenguajes del lado del servidor en páginas web," w3techs.com/technologies/overview/programming_language/all, Sep. 2014.

[30] "Bases de datos soportadas en PHP," http://php.net/manual/es/refs.database.php, Aug. 2014.

[31] "Sitio oficial de JSP de Oracle," http://www.oracle.com/technetwork/java/javaee/jsp/index.html, Aug. 2014.

[32] "Sitio oficial de JDK de Oracle," http://www.oracle.com/technetwork/java/javase/jdk-8-readme-2095712.html#docs, Aug. 2014.

[33] "Sitio oficial de JSE de Oracle," http://www.oracle.com/technetwork/java/javase/overview/index.html, Aug. 2014.

[34] "Sitio oficial de JRE de Oracle," http://www.oracle.com/technetwork/java/javase/jre-8-readme-2095710.html, Aug. 2014.

[35] "Tutorial de JavaBeans," http://docs.oracle.com/javase/tutorial/javabeans/, Aug. 2014.

[36] "Sitio oficial de Glassfish," https://glassfish.java.net/, Aug. 2014.

[37] "Sitio oficial de JBoss," http://www.jboss.org/, Aug. 2014.

[38] "Sitio oficial de Oracle," http://www.oracle.com, Aug. 2014.

[39] "Posicionamiento de mercado de las alternativas para páginas web," http://w3techs.com/technologies/market/programming_language, Sep. 2014.

[40] "Sitio oficial de Play! framework," https://www.playframework.com/, Aug. 2014.

[41] "Sitio oficial de Scala," http://www.scala-lang.org/, Aug. 2014.

[42] "Sitio oficial de YAML," http://www.yaml.org/, Aug. 2014.

[43] "Tutorial de JAR de Oracle," http://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html, Aug. 2014.

[44] "Sitio oficial de SBT," http://www.scala-sbt.org/, Sep. 2014.

[45] "Manejo de la cache en play," https://www.playframework.com/documentation/2.3.x/JavaCache, Aug. 2014.

[46] "Documentación de JSON," http://json.org/, Aug. 2014.

[47] "Sitio oficial akka," http://akka.io/, Aug. 2014.

[48] "Documentación Twirl de play framework," https://github.com/playframework/twirl, Aug. 2014.

[49] "Sitio oficial de LESS," http://lesscss.org/, Aug. 2014.

[50] "Sitio oficial de Ruby," https://www.ruby-lang.org/es/, Aug. 2014.

[51] "Sitio oficial de Coffescript," http://coffeescript.org/, Aug. 2014.

[52] "Sitio oficial de Hibernate," http://hibernate.org/, Aug. 2014.

[53] "Tutorial de JPQL," http://docs.oracle.com/javaee/6/tutorial/doc/bnbuf.html, Aug. 2014.

[54] "Documentación de Ebean," http://www.avaje.org/, Aug. 2014.

[55] "API para Java de Play para la versión 2.3," https://www.playframework.com/documentation/2.3.x/api/java/index.html, Aug. 2014.

[56] N. Leroux and S. Kaper, *Play for Java*. Manning Publications, 2014.

[57] "Página oficial de Matlab," http://www.mathworks.es/products/matlab/, Jul. 2012.

[58] "Sitio oficial de Backbone.js," http://backbonejs.org/, Aug. 2014.

[59] "HPCC de Amazon," http://aws.amazon.com/es/hpc/, Sep. 2014.